Sixth Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology (EPIC-6)

March 11, 2007

In conjunction with the IEEE/ACM International Symposium on Code Generation and Optimization,

San Jose, CA

WORKSHOP PROGRAM	
1:30-2:30	**Keynote** by *Don Soltis*, Senior Principal Engineer, Intel Corp.
2:30-3:00	Online Load Profiling and Dynamic Optimization in the Hotspot JVM. *Andrew Trick, Xiaoyi Guo, Richard Allen, Laurent Morichetti (HP)*
3:00-3:30	The Design and Architecture of MAQAOPROFILE: an Instrumentation MAQAO Module. *Lamia Djoudi, Denis Barthou, Olivier Tomaz, Andres Charif-Rubial, Jean-Thomas Acquaviva, William Jalby (Univ. de Versailles Saint-Quentin and CEA, France)*
3:30-4:00	Global Multi-Threaded Instruction Scheduling: Technique and Initial Results. *Guilherme Ottoni and David I. August (Princeton University)*
4:00-4:15	Break
4:15-5:15	**Keynote** "Itanium as a Horizontal Microcode Engine for Legacy Architecture Enablement" by *Ron Hilton*, Founder and Chief Technology Officer, Platform Solutions, Inc.
5:15-5:45	Optimal Placement of Fused Multiply-Add (FMA) Instructions. *Konstantin Serebryany (Intel Corp.)*
5:45-6:15	A Practical and Complete Implementation of SSUPRE without Static Single Use Representation. *Yao Shi, Tianwei Sheng, Hucheng Zhou, Dehao Chen, Wenguang Chen, Weimin Zheng (Tsinghua University), and Shinming Liu (HP)*

CO-CHAIRS

Rick Hank, HP
Sebastian Winkel, Intel

Online Load Profiling and Dynamic Optimization in the Hotspot* JVM

Hewlett-Packard Company

Andrew Trick
andrew.trick@hp.com
408-447-8862

Xiaoyi Guo
xiao-yi.guo@hp.com
408-873-6683

Richard Allen
richard.allen@hp.com
408-447-4568

Laurent Morichetti
laurent.morichetti@hp.com
408-447-2153

1. Abstract

Memory access is often the primary bottleneck for typical applications. On the Itanium platform, it is especially important for compiler to be sensitive to memory latency. Instruction scheduling for best-case load latency generally results in a high number of processor stall cycles caused by frequent accesses to the L2 data-cache, or higher levels of the memory hierarchy.

We have implemented a framework to address this problem within the Hotspot* Java Virtual Machine. The JVM uses the Itanium Performance Monitoring Unit (PMU) to profile the application while it is running. When Java methods are first compiled, no load profile data is available. However, if the online profiler reports that a compiled method suffers from frequency data cache misses, that method is recompiled using optimizations that reduce the penalty caused by delinquent loads.

The load profiling framework is designed to be viable as an out-of-box product feature. Consequently, we have given special attention to efficiency. Unlike previously published research, our framework is context sensitive: it processes the profile data without losing any available call chain information. We are in the process of using the load profiling framework to prototype profile-driven compiler optimizations, and we show preliminary results.

1. Load Profiling Overview

The online load profiler works alongside a dynamic compiler within the Java Virtual Machine. The dynamic compiler supports two levels of compilation, or tiers. The tier-one compilation of a Java method is optimized for speed, but does perform simple optimizations, including profile-driven optimization based on the interpreter's branch profile, such as inlining. While the tier-one generated code executes, the JVM collects a profile of the loads that are not serviced by the L1 data cache. If a tier-one compiled method experiences frequent d-cache misses, a tier-two compilation request is initiated. The tier-two compiler performs aggressive optimizations that generally increase compilation time, and may apply optimizations based on the load profile data. Although the PMU continues to sample loads from tier-two generated code, the profiler currently ignores these samples.

The PMU stores samples in a d-cache event address register (DEAR) that provides a load’s instruction address, data address, and access latency. The online profiler records these samples in hash table entries keyed on the instruction address. Each profile record contains two counters. The first counts the number of samples matching this instruction address. The second counts number of cycles corresponding to the sum of the sample latencies. Thus the average latency can be computed at any time.

The PMU samples identify each load by its instruction addresses within tier-one compiled code. The dynamic compiler associates these addresses with load instructions in tier-two compiled methods by correlating them with the Java bytecode index (bci) from which loads originate. Both compiler tiers may perform inlining, so we also maintain the context sensitivity of the samples by correlating the inlining call chains between the two tiers.

Bytecode-level information is maintained during tier-one compilation by tagging each instruction with a single integer ID that represents its extended bci (xbci). The xbci encodes the load's call chain within the tier-one compilation unit, and the bci within the bottom method of the call chain. A mechanism is built into the optimization framework for inheriting xbci’s during instruction transformation. At the end of tier-one compilation, the inlining decisions (inline tree) are stored alongside the compiled code. Each node of the inline tree contains a map of bci’s to instruction addresses for all the loads in that

method. We refer to these map entries as “load descriptors”, and we refer to their parent node as the “tier-one inline tree node”.

The tier-two compiler correlates load instructions with tier-one instruction addresses first by finding the tier-one inline tree node that best matches the current call chain within the tier-two compilation unit. It then consults the table of load descriptors to locate an address that matches current load’s bci. Once an instruction address is recovered, the compiler searches the profiling data for a matching hash table entry. If a profile record exists for the load, the load instruction is tagged with its average latency. The same framework used to propagate xbci values throughout tier-one compilation is reused to propagate load latencies during tier-two compilation.

2. PMU Sampling Overhead and Accuracy

The load profiler is designed to be an out-of-box product feature that should not exhibit noticeable overhead in any circumstance, especially considering that some applications will not benefit from load-profiling driven optimizations. The profiling overhead in terms of CPU-time can be expressed as follows:

```
   sample_cost := cycles / sample
   sample_freq := samples / cycle
overhead = sample_freq * sample_cost          (1)
```

The load profiler's sampling frequency is adjusted to meet a predetermined overhead requirement based on the known cost of sampling. The only slight complication is that the profiler configures the PMU to sample the DEAR every N d-cache misses, rather than every N cycles. This provides a more representative sampling of delinquent loads, but it means that we do not have direct control over sample frequency. However, we can easily estimate the sample frequency in real-time by timing the interval between sample collection and counting the number of samples collected:

```
sample_freq ~= sample_count / collection_interval     (2)
```

The runtime can then iteratively adjust the sample rate and reestimate sample frequency to achieve the desired target frequency.

The load profiler strives to maximize accuracy and coverage while minimizing runtime overhead. A short sampling period allows the runtime compiler to respond to optimization opportunities sooner. At the same time, we want to acquire a complete profile prior to recompilation. The solution to these tradeoffs lies in reducing the per-sample cost of profiling. As shown in formula (1), decreasing the sampling cost by 50% allows doubling the sampling frequency.

We have employed some important techniques to reduce both the per-sample cost and the data footprint of load profiling. First, we take advantage of the HPUX kernel's support for buffering PMU samples. Avoiding user-level signal handling for each sample is critical. Additionally, we use a two-level hash table to aggregate and filter samples. When the profiler thread processes the kernel's sampling buffer, it aggregates samples into profile records using a highly tuned, fixed sized, first-level hash table. This first-level table is only accessed by the profiler thread, so requires no synchronization. The profiler thread periodically scans the level-one hash table to promote any records with high miss frequency, and discard the rest. Promoted records are stored in a second-level hash table that is also read by the compiler threads. We show statistics that demonstrate the effectiveness of these techniques at reducing overhead.

3. Context-Sensitive Interpretation of PMU Data

The tier-one compiler uses an extended-bci (xbci) to encode both call chain and bci information in a single integer. This allows the instructions to be tagged with very low overhead and the xbci’s to be propagated efficiently. Furthermore, each node in the inline tree is associated with a range of xbci indices such that the antecedent-descendant relationships within the tree can be queried simply by comparing xbci values. Consequently, the call chain corresponding to any xbci value can be efficiently computed. Load instructions may move during optimization, so the call chain must be recovered purely from the load's xbci value in order to annotate the compiled code with load descriptors.

Before querying profile data during tier-two compilation, the compiler first finds the longest matching tier-one call-chain, because that will provide the most accurate load

profiling information. Once a call chain match is found, the instruction address can be obtained by searching the table of load descriptors within the tier-one inline tree node for a matching bci. The results of the call chain queries are cached within the tier-two inline tree, significantly reducing the amount of time spent searching for matches. The following algorithm is used to retrieve a tier-one inline tree node given the tier-two inlined call chain leading to a load instruction:

(1) If the tier-two inline tree node corresponding to the current call chain has already been matched, then reuse the result, which is either a match failure or a cached reference to the tier-one inline tree node. Otherwise, proceed to step (2).

(2) Recursively find the best match for the parent of the current tier-two inline tree. This matches the segment of the call chain above the bottom-most method. If the parent has a successful match, then search the parent's cached tier-one inline tree node for a child node at the same callee bci. If the callee bci exists, the best match is found. Otherwise, proceed to step (3).

(3) Search for tier-one compiled methods one at a time while walking down the tier-two call chain. The traversal does not start at the top of the call chain. Instead start just below the last unsuccessful match, which is simple to determine. If the current node's parent was not successfully matched, search only the tier-one method corresponding to the bottom-most method in the call chain. On the other hand, if the parent was successfully matched (but did not contain the current callee) then start just below the top of the parent's matched call-chain. For example, let the current call chain be A->B->C->D (the load instruction originates from method D). For clarity we omit the bci qualifier at each call site, although it is obviously essential for non-trivial inline trees. Now assume the tier-two inline node parent (A->B->C) best matches the tier-one inline node B->C, which was determined in step (2). We know that B->C->D does not exist in tier-one because step (2) already failed. The top of the parent's matching chain is B, so we begin searching for a match just below our parent's top at C. Therefore, we first check for the existence of C->D in tier one compiled code. If method C was not compiled, or did not inline D, then we check if method D was itself compiled.

Using this algorithm, a match is attempted at most once for each tier-two inline tree node. For any given match, the worst-case number of tier-one methods searched equals the depth of the call chain. However, an amortized constant number of tier-one methods are searched per tier-two node. Thus, ignoring the bci comparisons, the algorithm has linear complexity—no worse than a context insensitive approach.

Note that the matching tier-one inline tree node always corresponds to the method at the bottom of the current call chain. So at that point, finding the load's instruction address is simply a matter of comparing the bci values in the tier-one inline tree node's load descriptor table. This table is small because it is limited to the number of loads in the bytecode method prior to inlining, and it is sorted by bci, so it can be quick-searched.

Similarly, step two of the algorithm uses quick search within the table of call sites to locate a matching inlined callee. Comparing the call-site’s bci value is much more efficient than comparing method names or method pointers. (Storing method pointers in compiled code would have complicated implications for runtime memory management.) Although call-site bci is usually sufficient to identify a unique callee, it is possible for different methods to inlined at the same bci across tiers. For example, a virtual call site may be optimized differently in tier-two versus tier-one. Additionally, multiple methods may be inlined at the same call site during the same compilation, such as when a virtual call site has two frequent targets. Therefore, we annotate the inline tree with method bytecode size in addition to call-site bci to differentiate between callees at the same call site.

4. Load-Profile-Driven Optimizations

We plan to exploit the load profiling framework by introducing optimizations in the dynamic compiler that hide some of the latency of delinquent loads by overlapping them with each other and with other computation. For our purposes, we divide delinquent loads into three categories:

1. Stride-based access of large arrays
2. Pointer indirection from a “parent” object to its “child” in which a simple parent-to-child relationship exists
3. Any other non-spatially located accesses to the Java heap.

Our profile-driven optimization does not address the first category, stride-based access, for two reasons. First, it is not common in the application code for current benchmarks of interest. Note that special tasks like memory initialization and array copying are handled by hand-optimized routines. Second, most loops that perform array access have known stride, and can be prefetched based on static analysis.

We also do not specifically optimize access to objects with simple parent/child relationships. We consider the relationship between objects “simple” when the child object is referred to by single parent, and the parent and child are coallocated and have equivalent lifetimes. Such cases present obvious opportunities for exploiting spatial locality because the distance between the object can be controlled by runtime memory management.

Prior work based on the ORP JVM [2] has leveraged the PMU to detect these cases. In practice, the culprit is almost always a “parent” String object referring to its “child” character array. We believe the majority of these cases can be optimized without the PMU simply by the enhancing the JVM garbage collector and inserting prefetches based on static type information.

The purpose of our load profiling framework is to optimize loads that do not fall into the two categories above. For this reason, the compiler is unable to guess the address of a delinquent load prior to its being materialized in the instruction stream. Consequently, we focus on techniques that simply hoist the data flow chain leading to a delinquent load consumer as early as possible within a compilation unit. The following techniques are at our disposal:

- Instruction scheduler enhancements for average (as opposed to fixed) load latency: The dependence height of any delinquent load reported by the PMU is increased, while other loads remain at unit latency. This increases the distance between delinquent loads and their consumers without forcing unnecessary stop bits.
- Instruction scheduler support for clustering delinquent loads: This attempts to schedule multiple delinquent loads to separate cache lines before any the loads' consumers.

- Predicated load hoisting: The scheduler can allow loads to be scheduled prior to a control dependence (typically a null check) by predicating the load under the control determinant predicate.
- Predicated global code motion: As with predicated hoisting, loads are guarded by a control determinant predicate, but can be hoisted across scheduling regions—for example across loops.
- Straight-line prefetching: Prefetch instructions can be inserted for delinquent load addresses. The prefetch instructions will be exposed to both global code motion and superblock scheduling. This is simply an alternative to the above techniques, but is unlikely to produce additional benefit unless register pressure is high.

We will provide preliminary results from initial experiments using standard Java benchmarks.

5. Future and Related Work

The online load profiling implementation is complete. The optimizations mentioned in the previous section are already planned, and some prototypes have been developed, but they have not yet been tuned for performance. Future work includes adding optimizations to the compiler that are guided by the load profiling. In particular, we plan to use IA64 hardware support for unsafe control and data speculation by generating recovery code. Control speculation is a slight improvement over predicated load hoisting because it eliminates the dependence on the compare instruction's predicate. Data speculation is complimentary to the other technique because it can be employed when the load's address is available prior to a call site or potentially aliasing store instruction.

In the future we plan to collect more profiling information, particularly Branch Trace Buffer (BTB) samples. This will further improve profile-guided recompilation. Profiling execution frequency concurrent with d-cache miss frequency is potentially valuable because it allows us to roughly estimate miss ratio. The miss ratio of a load is the frequency that it misses in the d-cache divided by the load’s execution frequency. Currently, our profile information only provides miss frequency, which alone may be a poor guide for optimizations. For example, we may insert a prefetch for a load that usually hits the L1 d-cache, simply because the load was executed so frequently that,

even with a low miss ratio, the miss rate appeared frequent to the profiler. Such overly aggressive prefetching is likely to degrade performance.

A similar framework for load profiling was implemented using the BEA Jrockit* JVM for Itanium [2], and showed encouraging results. Our work validates their approach to profile-driven recompilation using an entirely different production JVM and compiler framework. We hope to supplement the previously published information by providing design details such as correlation of load samples to call chains, and by specifically measuring the effectiveness of various techniques for hoisting delinquent loads. We also evaluate the framework using the more recent SPECjbb2005 benchmark.

6. References

[1] Adl-Tabatabai, A., Hudson, R. L., Serrano, M. J., and Subramoney, S. Prefetch injection based on hardware monitoring and object metadata. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, PLDI'04*, Washington DC, June 2004, pp. 267-276.

[2] Greg Eastman, Shirish Aundhe, Robert Knight, and Robert Kasten, Intel. Dynamic Profile-Guided Optimization in the BEA JRockit(TM) JVM, In *3rd Workshop on Managed Runtime Environments*, MRE'05, San Jose, CA, March 2005.

The Design and Architecture of MAQAOPROFILE: an Instrumentation MAQAO Module

Lamia Djoudi[1], Denis Barthou[1], Olivier Tomaz[1],
Andres Charif-Rubial[1], Jean-Thomas Acquaviva[2] and William Jalby [1]

[1] Université de Versailles Saint-Quentin, France,
[2] CEA, France,
Tel: +33(0)1 39 25 43 43 - Fax +33 (0)1 39 25 40 57
Contact: Lamia.Djoudi@prism.uvsq.fr

Abstract

This paper presents MAQAOPROFILE, an instrumentation module of the Modular Assembly Quality Analyzer and Optimizer tool (MAQAO). MAQAOPROFILE builds a comprehensive and fine grain profile of the application by inserting assembly probes in the assembly codes. Using a structured view of the assembly, the instrumenter enables the profiling at different levels, and can target specific functions, basic blocks, loops or instructions in the assembly with a limited overhead.

Additionally to time profiling, the instrumentation we propose is also performing value profiling. Value profiling provides an observation of the application from an inner point of view, at the opposite of the traditional profile which remains solely behavioral. We present several analyses enabled by this instrumentation tool, such as the computation of the application hotpaths, the analysis of prefetch address streams and the distribution of loop trip count, as well as their timings. Moreover, we describe how profiling values combined to static analysis performance evaluation are used by the performance advisor of MAQAO to help end-user to tune his application.

1. Introduction

The evolution of technology and of processor architectures leads to machines with increasing theoretical peak performance. However, the real application performance obtained is usually far from the peak performance. The degradation may be caused by : the source code (from dependences for instance), the compiler (from optimization issues), the operating system (page allocation policy not adapted to the application for instance) or the hardware architecture (data accesses generating bank conflicts for instance). In order to identify first and then possibly overcome these degradations, research and development efforts are mobilized to develop tools and solutions in order to understand, predict and evaluate code performance.

Computer architects need tools to evaluate how programs will perform on new architectures. Software writers need tools to analyze their programs and identify critical pieces of code. Compiler writers often use such tools to find out how well their instruction scheduling or branch prediction algorithm is performing or to provide input

for profile-driven optimizations. Program analysis tools are extremely important for understanding program behavior.

Most of the performance analysis tools/toolkits are among two main classes. The first one is focused on the exploitation of hardware performance counters while the second relies on code instrumentation or even transformation. Traditionally, hardware counters, profiling information, static analyses and even expert knowledge are exploited individually or at best interconnected through *ad-hoc* tools. As a response to the enlarging gap between needs and existing software, we have developed a Modular Assembly Quality Analyzer and Optimizer (MAQAO)[1]. The concept is to centralize all low level performance information and by finding correlations, produce performance tuning advices. As a result, MAQAO produces better results than the sum of the existing individual methods. MAQAO is designed as a set of interlinked modules each of them being loosely coupled to the others.

MAQAO is working at assembly level: Assembly code is a gold mine of information. This information just need to be correlated with profile information or with performance evaluations to deliver valuable tuning feedback (either brought back to the user or given to the compilation chain). Additionally, being based after the compilation phase allows a precise diagnosis of compiler optimization successes and/or failures, diagnosis that can be reported back to user for potential improvement via compiler flags or source code modifications. As an automatic tool, MAQAO processes large amount of data and applies optimizations and diagnosis to the whole code. It is restricted to a limited number of hot spots but can study a complete application. This approach allows to track down most of the little percentage loss all over the code. Finally, we rely on existing standard solutions when they are effective: hardware counters are supported through perfmon [2], data storage is handled with a database (which can queried by SQL) and a scripting language is embedded within MAQAO to allow user to extend it according to his own needs.

Our purpose is not to design yet another software tool but to implement an optimization methodology and to propose a real Performance Framework. Assembly code inspection is done statically, data profiling is done using code instrumentation, and hardware counters fit their traditional role of hotspot detection. Compared to other existing instrumentation tools such as Pin [24], MAQAOPROFILE introduces some unique features and analyses, based on the profile of inner loops, hotpaths and prefetch address streams.

This paper presents MAQAOPROFILE, an instrumentation module in MAQAO tool. A complement module to MAQAOPROFILE is MAQAOADVISOR. It is a performance advisor driving the optimization process through assembly code analysis and performance evaluation. The rest of this paper is organized as follows. In

section 2, we outline the MAQAO framework. Section 3 provides the overall design of MAQAOPROFILE. Section 4 describes the architecture and features of MAQAOPROFILE. In section 6, we describe how information collected by MAQAOPROFILE is used by MAQAOADVISOR with information collected by a static analysis of the code. In section 7, we point the reader to related work in the area of instrumentation tools. Finally, in section 8, we conclude and propose some future works.

1.1 Motivating Example: Hardware Counters Are Not The Panacea

Performance is often a multi-dimensional problem, this is obviously due to the number of actors involved and their respective complexity.

Despite the multi-faceted nature of performance analysis, the current trend is to rely heavily on hardware counters. Results obtained by this approach are numerous and of high quality, therefore counters are now ubiquitous on every processor hitting the market. Additionally counters are more or less standardized mainly due to the PAPI [3] initiative. However, while being helpful this uni-dimensional method is not sufficient and often answers only to a part of the performance question. For instance it is well known that what is important *is not the number of cache misses, but if these misses were overlapped or not*. In fact, a large amount of the hardware budget is already spent for latency tolerance techniques (more than 96.3% of the 1.72 Billion transistors on Montecito processor [4] if caches are considered as belonging to such techniques), therefore in term of efficiency these features should be used, but performance trouble appears when they are overused.

Obviously the miss cost problem could be tracked down with a thorough performance counter analysis coupled with a large set of experiments: however the cost to pay for that shows that counters are not the adequate tool. The following example, coming from a real world optimization problem, emphasizes hardware counters power and limitation.

Optimizing FFTW kernels

FFTW (Fastest Fourier Transform of the West) [5] is a highly tuned version of the FFT. FFTW is built over decomposition of complex FFT in simple computational blocks named *Codelets*. These codelets, which are small pieces of code, contain most of the computation. A generator is able to produce automatically different flavors of codelets but all of the codelets are generated in C, thus performance depends on the compiler. Trying to push performance beyond the best found compiler options (`-O3 -fno-alias`), starts with a close examination of hardware counters.

Loop trip	2	4	8	16	32	64
CPU Cycle	87	150	278	548	1065	2101
Inst. issued	304	542	1018	1970	3874	7682
F. Ops issued	68	136	272	544	1088	2176
Stall Cycle	18	36	72	157	307	607
Stall %	21%	24%	26%	29%	29%	29%

Table 1. Hardware counter measurements for FFTW 4 codelet. Stall cycle is fairly limited, and IPC seems satisfying. Overall, performance is around 33 CPU cycles per iteration.

Loop trip	2	4	8	16	32	64
FP. instr.	68	136	272	544	1088	2176
MEM instr.	44	88	176	352	704	1408
Bound (cycles)	28	56	112	224	448	896

Table 2. Corresponding Essential Instructions determined by MAQAO static analysis. Code appears to be compute bound with an optimal execution time of 14 cycles per cycles.

All the experimental results reported in this paper were obtained with a state of the art compiler, *icc* v9.0. Experiments were run on a 1.6 GHz / 9 MB L3 Itanium 2 system.

Essential instructions: determining intrinsic code bounds

Results from Table 1 depict dynamic results, i.e. a code with good IPC and a rather limited fraction of stall cycles. Dynamic and static results match for the number of instructions issued: 304 instructions are issued in two iterations, 542 for four iterations, corresponding to an average of 120 instructions issued per iteration plus 64 instructions of overhead (the same applies for cpu cycles). The dynamic measurement is around 33 cycles per iterations.

Stalls are determined statically as the difference between the instructions issues and the cycles prediction provided by the compiler. In our case (simple code without branch), stalls are stemming mostly from dependencies (load to use latencies, floating point latencies ...). The compiler predicts 5 stall cycles per iteration while the hardware counter measured 9 stall cycles per iteration. Adding this 4 extra cycles to the number of cycles estimated by the compiler, gives 32 cycles per iteration which is very close to the 33 cycles measured. Therefore, optimizing the code by reducing the number of stalls would bring at best a 20% speed-up.

The performance issue is somewhere else: MAQAO reports key information on loop structure: loop is neither pipelined nor unrolled. MAQAO also collects static metrics on the assembly code and lists the following essential instructions:

34 Floating point operations [1]:

$$\left.\begin{array}{ll} 6 \text{ FMA} & -2 \text{ per cycle} \\ 22 \text{ simple flops} & -2 \text{ per cycle} \end{array}\right\} 14 \text{ cycles}$$

22 Memory operations:

$$\left.\begin{array}{ll} 14 \text{ loads} & -4 \text{ per cycle} \\ 8 \text{ stores} & -2 \text{ per cycle} \end{array}\right\} 8 \text{ cycles}$$

If we consider a perfect overlap between memory and computational operations (data dependencies completely concealed), bound is $\max(memory, floating\ point)$. Therefore this loop is compute bound: 14 cycle per iteration. Table 2 allows a quick comparison with static analysis results.

While dynamic measurements return a cost of 33 cycles per iteration, the optimal static schedule is 14 cycles. This exposes the code bloating problem: among all the issued instructions, how many are *essential/useful* instructions ?

Monitoring dynamically the stream of addresses manipulated by the loop reveals that addresses appear in sequential order. This means that iterations are not interleaved nor overlapped. Therefore performance is either constrainted by strong data dependencies or parallelism is weakly exploited. Simple analysis of the source code indicates that the iterations are independent. However, the source code contains read and write array accesses of the form $A(1)$, $A(ios)$. Not knowing that ios is strictly positive forces the compiler to have a very conservative schedule. Following these deductions, using the classic versioning (or specialization) optimization for the given loop trip allows to provide the compiler with the critical information that ios value is never set to 0. This time, the compiler generates a much more efficient code, performance comparisons are provided in figure 1, where the optimized version delivers a speed-up of 40% as soon as iterations are over 8. To sum up, a tool assessing assembly code quality is essential to produce high-performance code. Indeed, even state-of-the-art compilers do generate poor quality assembly from real codes and this is difficult to evaluate using a pure dynamic approach. A stage of static analysis delivers interesting results, and is a shortcut to optimizations w.r.t. to the long and burdensome hardware counters analysis.

2. The MAQAO Framework

For all our experiments and analyzes, we resort to MAQAO. MAQAO combines static and dynamic analysis of assembly codes, on the Itanium platform, in order to exhibit any shortcomings in the compiler optimization process and correlate dynamic behavior with static analysis in order to improve code efficiency. The module of

[1] One FMA counts as 2 floating point operations.

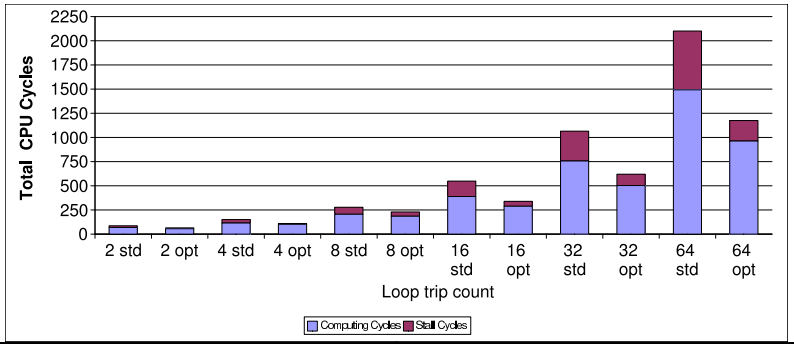


Figure 1. FFTW4 Codelet Performance. CPU cycles are displayed for the standard (*std*) and (*Opt*) which is the version after code specialization. Performance is down from 33 cycles per iteration to 18 cycles per iteration. The fundamental aspect is that the observed improvement is greater than the number of originally measured stall cycles (in red). Thanks to non essential code elimination. Hardware counters are blindly reporting computing cycles (in blue) while the CPU is processing useless instructions.

importance for the methods presented in this paper are:

Static analysis: Takes the assembly code as input, parses it and extracts code structure in the form of control flow graph, call graph and data dependence graph.

Dynamic analysis: MAQAO injects assembly probes at various parts of the code. At execution time these probes record time stamp and also data manipulated by the code. Profiling information is used to build an execution summary. The probes can be accessed by end-user or used by the Oracle module.

Performance advisor: It provides a thorough guidance to help the code tuning process by providing a synthesis of information collected by preceding phases. This advisor helps the user to take high-level decisions, such as selecting specific compilation flags or splitting the code and compile different parts with different flags.

Optimization: The optimization can be done by modifying the instruction code, the data layout or any other element that is involved in the execution of a program. Our optimization methods are based on: (1)modifying assembly code,(2)rescheduling assembly code (e.g. peeling), (3)merging different versions (e.g. preftch/noprefetch).

Modules are centered around a core module and a database which is ultimately the place where every piece of information is stored. Database is in charge of ensuring data persistence and also offers a standardized storage format.

The MAQAO framework supports the following features: It allows a user to view a low-level graphical representation of the code. It analyzes the assembly code generated by compiler (icc and gcc on IA32 and IA64). It can give to the user all necessary information for fine-grain tuning (e.g. pipelined loop, resource usage). A user can instrument the assembly code. Finally MAQAO provides an expert system to help user to deal with complex architecture and to understand (1)optimization failures, (2) obscure compiler decision and (3) propose effective optimizations. As detailed in figure 2, data are displayed in a GUI to offer a way for end-users to navigate among all the amount of code information. Still to leverage end-user effort, MAQAO provides a LUA [16] interface to access most its internal data structures. A user can write, based on this API, his own requests which fit to their particular problems.

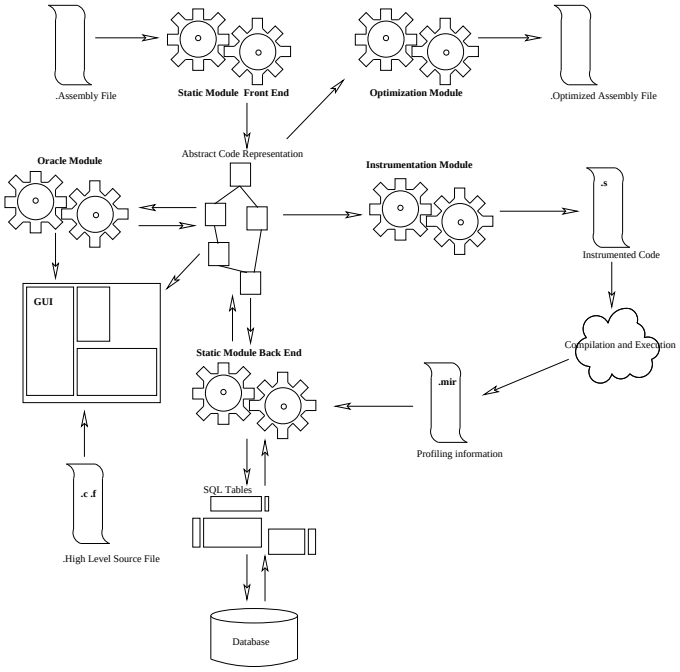


Figure 2. MAQAO components and module organization. The backbone is the abstract code representation, a set of data structure for program performance information. Data persistence across MAQAO executions is ensured by a database.

3. Overall Design of MAQAOPROFILE

MAQAOPROFILE consists of four componenents, as depicted in figure 3.

At first MAQAO takes as input assembly files generated by the compiler and parses them to produce a structured representation of the assembly code. The instrumentation follows the different steps described thereafter:

1. *Instrumentation:*the user selects the appropriate level to instrument. MAQAOPROFILE injects a limited number of extra-instruction, named *assembly probes*, around the targeted code fragments to monitor. This low-level scheme is technically challenging to implement (e.g. ensuring the integrity of the register stack for IA64 code) but it allows minimal interferences with both the behavior and performance of instrumented applications.

2. *Instrumented executable generation:* this step generates the modified executable code. A number of additional files are likewise generated, as well as a `makefile`. The application main function is wrapped with a new function defining necessary instrumentation structures so that the complexity of application compilation process remains unchanged.

3. *Execution:* the code execution generates a script file (in LUA language) that contains both the results of the instrumentation and the program to store these results. Execution of this script file fills MAQAO database with the appropriate information. MAQAO then combines static and dynamic performance evaluations and can deliver answers to user queries.

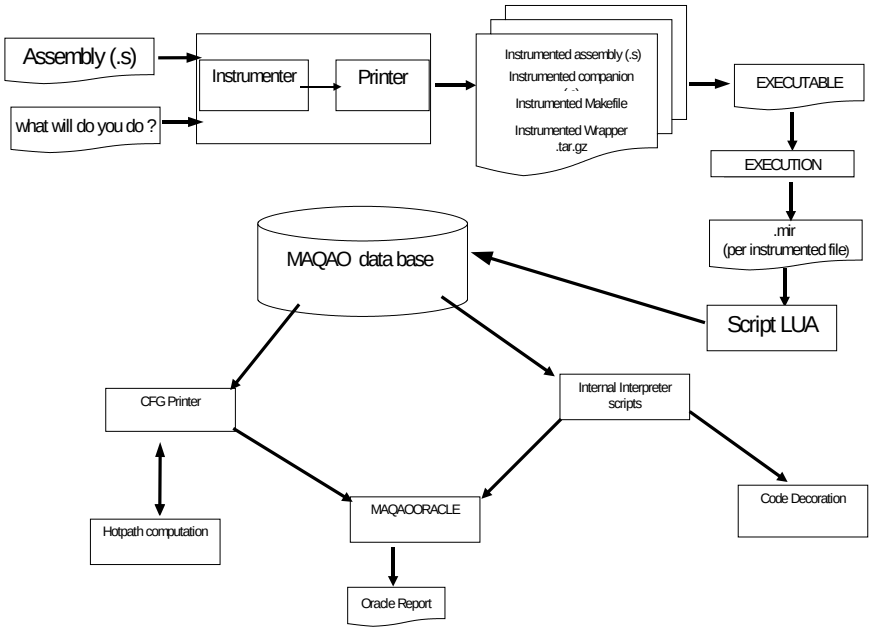


Figure 3. The MAQAOPROFILE process

The information resulting from the instrumentation is :

- Posted on the Control Flow Graph. The CFG printer is used to generate hotpath computation. Hotpath results are reloaded from data bases when reloading an assembly for which this profiling was performed.
- Summarized by the performance advisor.
- Summarized in new windows of MAQAO interface.

The two previous points are generated with LUA scripts and can be changed according to user needs.

4. Detailed Features of MAQAOPROFILE

MAQAO proceeds to code instrumentation automatically. This is done by injecting a limited number of extra-bundles, named *assembly probes*, around the targeted code fragments to monitor. These bundles are in charge of storing some specific registers in a dedicated memory zone. MAQAO ensures that the register stack remained unchanged by the instrumentation (by analyzing allocated registers or by adding spill/fill instructions). This low level technique has several advantages: it does not alter the behavior of the compiler, neither for the code generation nor for the code optimization, since the instrumentation is done as a post-compilation transformation, and the run-time overhead is kept small. Moreover, it is possible to instrument code at the assembly level, that does not correspond to any source code. For instance, the compiler can generate many different versions for one loop. Computing the loop trip count distribution of the different generated loop versions is only possible at the assembly level. Likewise, profiling the behavior of an inlined function in a specific call site is only possible at the assembly level.

MAQAO instrumentation can be done at function level, loop level, basic block level or instruction level. For all these levels of instrumentation MAQAO monitors and stores the value of the clock register and builds execution time profile. Additionally MAQAO is able to store every register value manipulated by the code and dynamically process it. Monitoring values manipulated by a binary at run time is often reffered to as *value profiling* [9]. Value profiling is one of the key features of MAQAOPROFILE, and it is often the missing link between the observed behavior on the hardware and the nature of the application. This feature yields to numerous optimization opportunities.

4.1 Block level profiling: Hot Paths

For instrumentation done at basic block level, we have found only one valuable usage: Hot path computation. Therefore we specialize this level, and basic block instrumentation is dedicated to production of hot paths. While classic tools such as `gprof` can isolate the most important function, its scope is too narrow, limited to almost the notion of function. Identifying the path at run-time which crosses the whole program where the application spends the most of its time is a key element for understanding application behavior [10], [11].

Hotpath is based on the classic Bellman algorithm (BA) [12]. Basically hot path computation is done in the following steps:

1. Assembly probes are injected at the entry/exit of every basic blocks. Thus at run-time, a counter is incremented each time the corresponding edge of the CFG is taken.

2. After the execution stage, once all counters are available, the interprocedural control flow graph is computed (taking the union of control flow graphs for all functions). Back edges for loops are deleted to avoid to trap the Bellman algorithm and to save computation time as BA's complexity could approach $O(n^3)$ [2].

3. Once the CFG for the whole code is produced with all edges annotated by their respective counter, edges are weighed correspondingly to the edge between the calling block and the return block. These edges are then deleted to force Bellman algorithm to go through the function. Then Bellman algorithm is used to update the evaluation of the vertex (and not edges). An extension will be to evaluate vertex, for instance according to the block latency in order to compute the most expensive path, or to the number of instructions to compute the largest path, but currently they are set to 1 as we focus on frequently used paths.

   The first basic block of each function has an evaluation set to 0, this prevents the algorithm to stumble on call sites. Therefore an understated assumption is that a function could be considered as an edge with an evaluation equal to the sum of the evaluation of its hot path (like introducing a delay due to its hot path). This can be seen as a recursive hot path computation.

4. Finally the critical path is found. Taking advantage of this stage, multiple-exit loops are examined and graph exploration is pushed up to the point where the same edge is never taken twice. Result of this procedure is depicted in Figure 4.

[2] Notice that currently our hot path algorithm does not support recursive function calls.

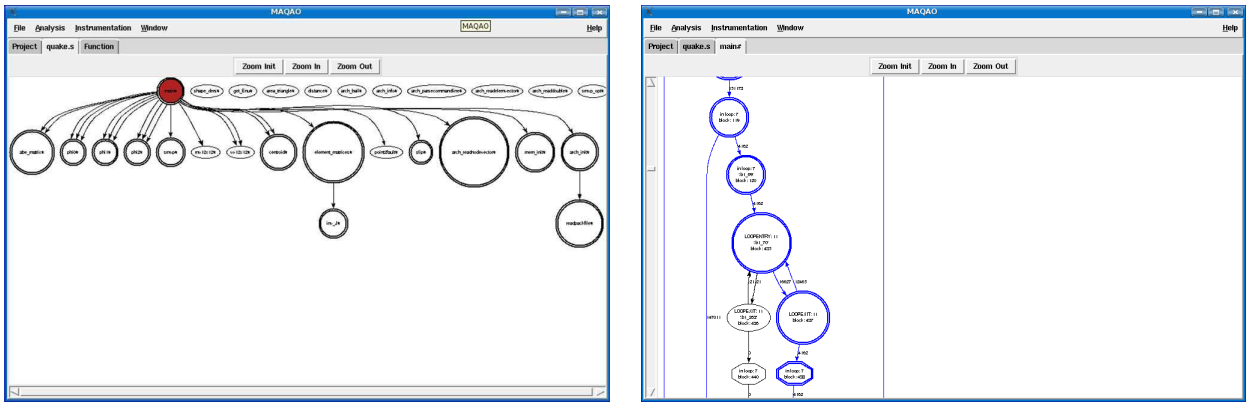
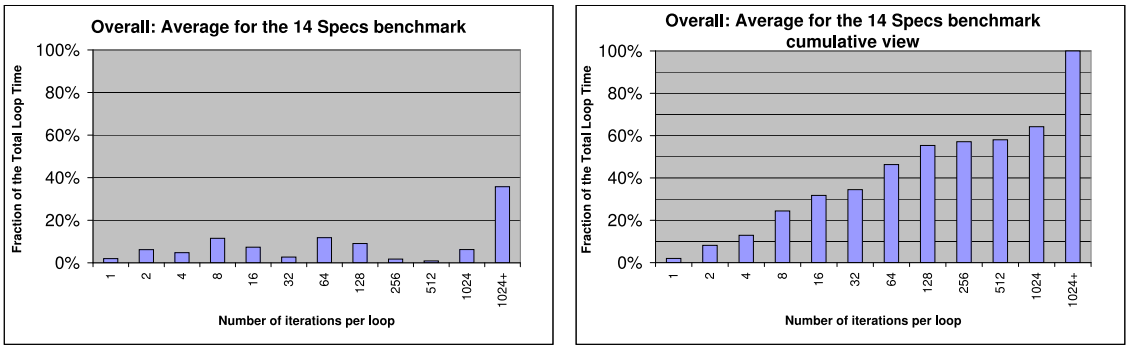


Figure 4. Displaying Hotpath information in MAQAO for 183.equake CG and a sample of its CFG.



(a) SPEC FP overall loop distribution

(b) SPEC FP overall cumulative loop distribution

Figure 5. Loop Execution Time depending on loops number of iterations. These graphs depict the execution time distribution per loop depending on the total number of iterations. Histograms summarize iteration weight for an interval in power of two:]0,1],]1,2],]2,4],]4,8] and so on. For instance the bar labeled 64 coalesces all loops with a total number of iterations between]33,64]. These two figures summarize results obtained on the whole SPEC FP benchmarks suite. 5(a) depicts the average on a per benchmark basis, i.e. each benchmark is considered individually without weight related to its execution time. 5(b) presents the same data but in a cumulative histogram which is convenient to grab the slope of loop distribution. From these two graphs it can be said than Spec benchmarks spend 25% of their loop time within loops of less than 8 iterations.

4.2 Loop Level Profiling

Figure 5 details results gathered by instrumentation at loop level of the SPEC FP 2000 benchmarks. We use the compilation flags (`-fast`, `-fno-alias` and so on) as reported by vendors on Spec website [13] except that inter-procedural optimization were disabled (`no-ipo`).

These numbers provide insight about code specialization opportunities for short loops where software pipeline and other unrolling techniques are often low-performers. In fact, this is specially important since low-level optimization are always targeting the asymptotic performance, neglecting all start-up effects as cold start. For a short trip, start-up cost can be the dominant one.

From a performance analysis point of view, these numbers are taking advantage of both time and value profiling. Time profiling allows to give a precise weight to all executed loops, therefore underscoring hotspot. Value profiling monitors the iteration count. Correlating this information provides the relevant metric: i.e. which hot loops are short. This a clear illustration of the interest of centralized approach for performance analysis.

4.3 Instruction Level Profiling

Knowing dispersion of values manipulated by each instruction can be valuable to take optimization related decision. For instance, characterizing address streams allows to detect bank conflict or aliasing problem. Observing that a given value is almost always constant can justify the cost of high level code specialization.

MAQAO currently supports the following rules: automatic detection of prefetch distance.

For a prefetch distance estimation, the algorithm is straightforward: all the instructions within the loop are instrumented. Data are stored in order sensitive way (rotating buffer).

1. Considering the address flow from the prefetch instruction: `0xf000 0xe000 0xf010 0xe010`
2. Considering the address flow from a load instruction: `0xef00 0xef10 0xef20 0xef30 0xef40 0xef50 0xef60 0xef70 0xef80 0xef90 0xefa0 0xefb0 0xefc0 0xefd0 0xefe0 0xeff0 0xf000`
3. Proceed to pattern matching for the first prefetch address. Since both instructions are within the same loop they are executed jointly (we do not handle case with if conversion). The first matching address in the load address flow (if any) returns the prefech distance. For instance, here data are prefetched 16 iterations ahead.

Evaluation of the prefetch distance can also be done statically based on a dependence graph (DDG). However, at the opposite of the static module, instruction instrumentation handles data stream interleaving (an optimization used Itanium by *icc* [14] to fetch alternatively two data streams with a single prefetch instruction)

4.4 Instrumentation Data Storage

Once the technical challenge of gathering data dynamically has been fulfilled, an interesting question is how to extract meanings from this vast amount of information. MAQAO supports two forms for data storage:

- *dispersion sensitive*: any new data are dynamically compared to the previously stored data. From this comparison data are put in the corresponding histogram bucket. For implementation reason bucket sizes are powers of two. This allows to build an accurate picture of data dispersion which is crucial to apply specialization techniques.
- *order sensitive*: the previously described method implicitly discards part of the dynamic information. In fact, building histogram drops the notion of sequence and execution order. Therefore, the second storage method is a rotating buffer where data are stored in the order in which they appear. This is very important for instance to computed address variation and to detect stride memory accesses.

5. MAQAORACLE: Performance Advisor

Gathering data and statistics is necessary for a performance tool, but it remains only a preliminary stage. The most important step is to build a comprehensive summary for end-user and extract manageable information. It is possible to interpret performance data in numerous ways... and to be lost. The *Performance Advisor* is built over a set of rules and metrics and act as an expert system to drive user attention within the performance landscape. Providing an expert system to help the user to deal with complex architecture was done by CRAY's AutoTasking Expert [15]. However ATExpert was focused on parallelization issue and was neither as extensible nor as sophisticated as MAQAO's performance module. Likewise, ICC outputs an optimization report, providing information on the success/failure of each optimization phase. This report can be considered as an execution trace of the optimization process. The report neither gives any clue on how to cope with some optimization issues, nor points out what are these issues. It just plainly states which optimization took place, where, and with which value of parameter. The goal is therefore different from MAQAO.

We first present the static analysis performed by MAQAO on the assembly code and then describe how both static and dynamic results are combined.

5.1 Static Analysis

Close inspection of assembly code is real mine of information. MAQAO allows to do it in a systematic and structured way. As a matter of fact, manipulating and understanding flat code is difficult. The lack of relief and hierarchy implies an important effort to filter out essential pieces from low interest part of the code. As an analyzer MAQAO static module extracts code structure. The structure is expressed through a set of graphs: Call Graph (CG), Control Flow Graph (CFG) and Data Dependencies Graph (DDG). These graphs are simple yet powerful tools to analyze a code.

For MAQAO a fundamental granularity is the innermost loop. Innermost loops constitute critical code fragment since they are the source of large fraction of the execution time. Most of MAQAO's analyses target this code granularity. The following metrics are computed for each inner loop:

- *Issue cost per iteration:* This metric is an over-estimation of the schedule latency. Basically it reports the number of cycles needed to emit all loop instructions. For this, only resources are taken into consideration. This metric allows to evaluate the cost of data dependences for the loop. A large gap induced by data dependencies hints that the loop should be unrolled more aggressively or targeted by other techniques to increase the available parallelism (loop fusion, hoisting and so on).

- *Cycle cost per iteration:* This is directly extracted from comments left by *icc* in the code. If needed it can be computed on the base of the instruction specifications [6], [7]. The cost is expressed as a function of the number of iterations. For non-pipelined loop it is simply in the form of: $a \times N$. N being the number of iterations (in the assembly). This static cycle evaluation is the reference point to estimate the effectiveness of dynamic performance.

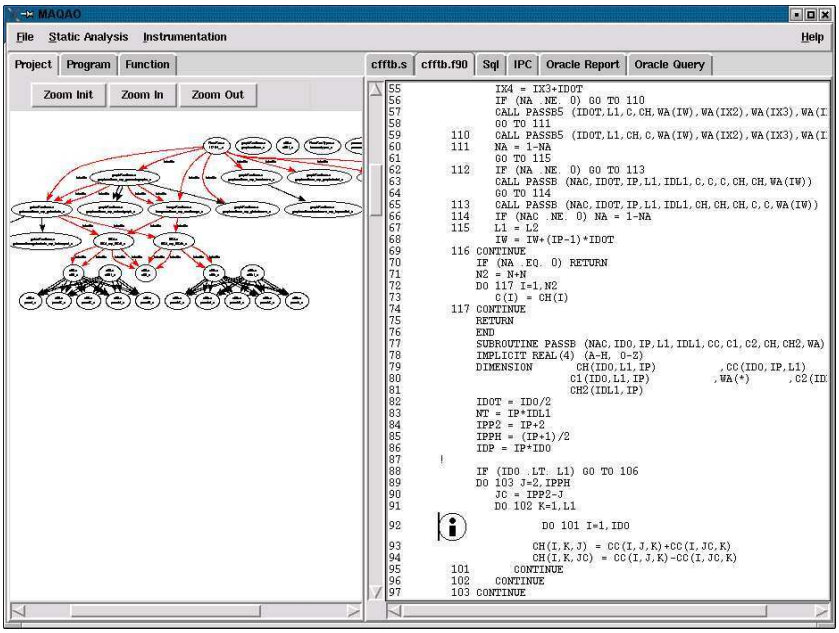


Figure 6. SPEC FP 2000 benchmark 193.Facerec scrutinized with MAQAO. On the left pane, the static call graph is computed. Red arrows represent call across different source files, while black arrows function calls within the same source file. Here all the files used to compile the benchmark are displayed, the amount of Red arrows is a straightforward metric to estimate inlining opportunities. Notice on the right tab, where the source code of one file is displayed, a ⓘ which indicates that on a single mouse click MAQAO displays high level analysis for this loop.

- *Theoretical cycle bound per iteration:* This metric indicates if loop is of compute or memory bound [8]. As introduced in section 1.1 by the motivating example, at some point it is important to know the ratio of important instructions compared to 'syntactic sugar' code. A well written code should exhibit a bound close to the issue cost and close to the cycle count. Knowing whether a loop is compute or memory bound is a powerful indicator of the kind of optimization techniques to use. Typically compute bound loops imply that many cycles are available to tolerate memory latency problems.

- *Pipeline depth:* Warming up and draining a deep pipeline is costly, and this can be overpriced in case of limited number of iterations. A pipeline loop has a different cost function than a regular loop. To achieve at least the equivalent of the first full iteration of the source code, a pipeline loop needs to execute as many iterations as the pipeline contains stages. Thus, there is a warming-up cost for pipelined loops (which should be paid off by a better throughput when the number of iterations increases). The cost function is: $a \times N + b$. N being the number of iterations, a the cost per iteration and b the filling-up/draining pipeline cost.

5.2 Hierarchical Reporting Approach

A classic pitfall for reporting tools is to overload end-user under a mountain of data. Therefore, this trend leads to miss the initial goal which is to help in the decision process. The needs of end-user differ, depending on which *level the decision*

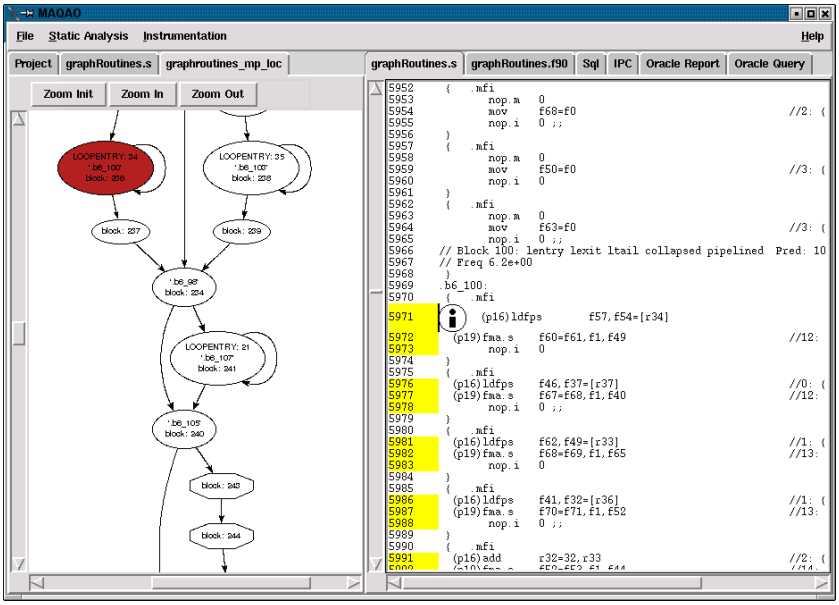


Figure 7. SPEC FP 2000 benchmark 193.Facerec: close inspection of function `graphroutine_localmove` with MAQAO. Part on the CFG is displayed on the left pane. Loops appear clearly with their back-edge. Hexagonal blocks (at the bottom) corresponds to basic blocks including function call. The assembly code matching the selected loop (in red) is highlight in yellow on the right pane. ⓘ indicates that on a single mouse click MAQAO displays low level analysis on this loop detailed in figure 8.

is going to be made: is it to chose between two compilers ? To select different compilation flags for the whole application ? To tune specifically a given loop? Being aware of this, MAQAO organizes information hierarchically. Each level of the hierarchy is suitable for a given level of decision to be taken: complete loop characterization, loop performance analysis, function analysis or whole code analysis. Additionnaly a filter can be set depending on the degree of confidence of Advisor answer or the potential performance gain involved.

- The first and most exhaustive level is the instructions level view. For each loop, selected instructions counts and built-in metrics are displayed. These counts require some knowledge to be interpreted but they represent the exact and complete input of what MAQAO is going to process in the upper stages. Instruction are coalesced per family (such as integer arithmetic, loads instructions and so on) and counted on a per basic block basis. However the goal is not to catch dispersion rules (hence the taxonomy) of the architecture, but to detail instructions that have been determined as being of special interest. This instruction count is enriched by built-in metrics : *Cycle cost per iteration*, *issue cost per iteration*, *Theoretical cycle bounds per iteration*. Together counts and metrics are exploited by Performance Advisor rules. Rules also process results gathered during application execution (instrumentation, hardware counter, cycle count).

- The second level, which is already an abstraction layer, only reports loop where some important performance features are detected, thus is filtering out a large amount of non-essential data. Additionally results are reported in a user-friendly way (not just a bunch of number but clear sentences!)

- The third and fourth levels are summary tables. Whereas respectively, for each routine and the whole code, a report counting the number of detected performance issues. Reading these tables is quick and was designed to ease comparison.

5.3 Example of Advisor Output

Here is an example of Performance Advisor output for a loop: Corresponding source code is:

$$\left.\begin{array}{ll} DO & I = 1, NPSTACK \\ & ZZ(I,K) = 0 \\ END & DO \end{array}\right\} \text{ should be replaced by a call to } \texttt{memset}$$

Results Sample [code M - function Chociso]:

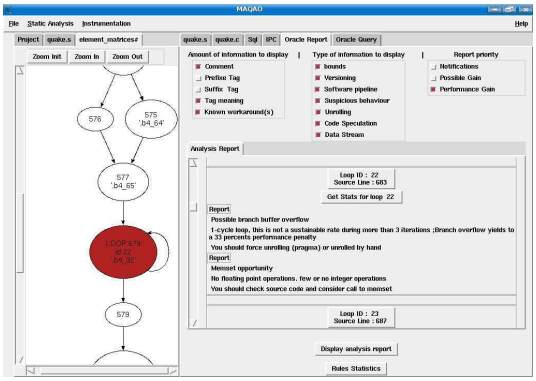
```
=> LOOP SOURCE LINE  426  ASSEMBLY VERSION 2 [MAQAOAdvisor internal loop. id  265 ]
     MAQAO report: No floating point operations. few or no integer operation.
     check source code and consider call to memset.
     MAQAO report: Scheduling is matching optimal bound.
     This is a clear sign of code quality but not a proof (beware of code bloating).
     MAQAO report: Analysis hints unroll factor of 8
```

Performance Advisor results are displayed in the MAQAO interface. In front of each loop of the source code, ⓘ gives access to the information computed by the Advisor concerning this loop. Note that several ⓘ in front of the same loop means that the function of the loop has been inlined several times. In interactive mode MAQAO can give different advices according to the inlined version, since the compiler may have changed the optimizations according to the inlining site.

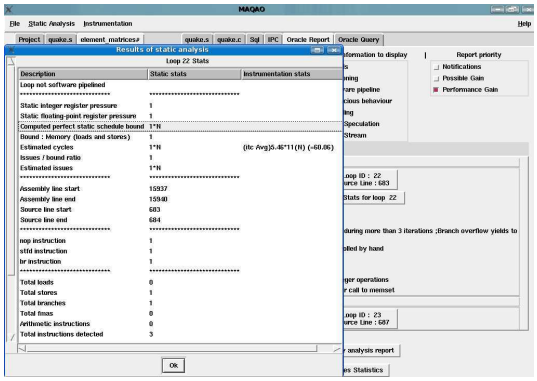
6. Related Work

Two main classes of low level instrumentation tools can be related to MAQAO. One class is consists of performance analysis tools aiming at understanding application behavior based on hardware counters. Fall in this category tools such as VTune (self-contained program), or PAPI (user-independent). The other family of tools is more focused on code manipulation like Salto, or code instrumentation such as ATOM or PIN. However MAQAO broad approach is more related to the path chosen by HPCview or to Finesse, the later being more focused on parallelization than code optimization.

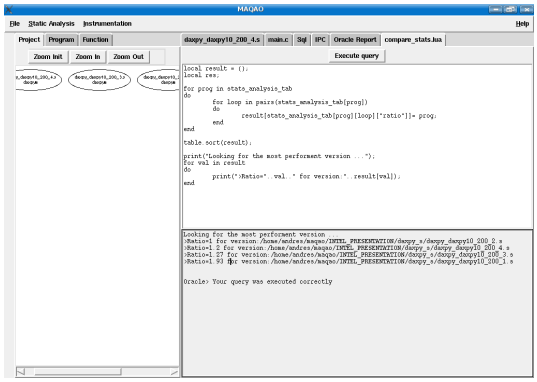
Hardware counters are extremely helpful for performance tuning, they are the backbone of tools such as `VTune`[17], `Caliper` [18] or Kojak [19]. Their usage is so widespread that an API gets standardized to describe their access [3]. Nevertheless, hardware counters are limited to the description of the dynamic behavior of an application and this picture needs to be correlated with other metrics. For instance, from the hardware counter point of view, code bloating (or even dead code) filling up functional units and leading to high IPC is seen as a desirable behavior.



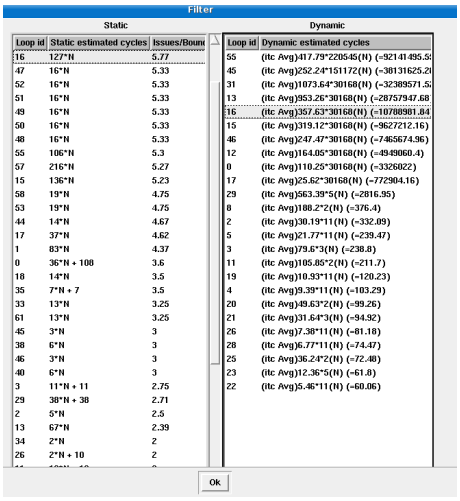
(a) Loop selected in the CFG: Performance Advisor displays analysis. The loop is an one-cycle loop and is similar to a memset routine. Thus it should be replaced by a call to the hopefully optimized memset routine



(b) Analysis of the loop with inclusion of dynamic information. Performance is bounded by Memory. While the perfect static schedule is estimated to 1 cycle per Iteration instrumentation results show that the average loop trip count is 11 and the cost 5.5 cycle per iteration. Static estimation is exceeded by a factor of 5.



(c) Writing a rule in LUA to analyze the whole code behavior. One may be interested in knowing 1/ the hot loops and 2/ if hot loops are far from their static schedule. Such request can be easily expressed in LUA. This language embedded in MAQAO allows to access all the internal data structure which are a repository of code performance information.



(d) Rule result: full comparison with static and dynamic performance including hot loops profiling information. Loops are sorted by decreasing weight and dynamic and static performance are compared. From a pure static evaluation loop 16 as a static schedule of 127 cycles per iteration which is 5 times the value of its optimal bound. Additionally it appears that dynamic cost per iteration is 357 cycles!. Clearly this loop worth investigation.

Figure 8. Example of Advisor analysis for a selected loop from the 183.equake CFG.

On the static side, Salto [20] is a framework dedicated to the implementation of complex assembly code transformations. The assembly is first parsed and then it is seen as a collection of C++ objects plugged into a user-developed application. Salto is more a toolkit than a tool and could appear as a back-end of MAQAO diagnosis chain: once a problem is identified by MAQAO, some transformations have to be applied by SALTO to solve this problem. DPCL [21] (Dynamic Probe Class Library) is a set of C++ classes from IBM originally based on Dyninst [22]. The purpose is to help developers to support dynamic instrumentation of parallel jobs. Probes can be inserted in a running binary to check the hardware counters or cycles for any function of the monitored code. Even if dynamic instrumentation is very appealing, DPCL does not include any notion of code inspection.

ATOM [23] (for Alpha processors) and Pin (for Intel processors) [24] instrument assembly codes (or even binary for Pin) in a way that instrumented specific instructions are executed trigger the execution of user-defined routines. For a comparison with MAQAOPROFILE, Pin can perform instrumentation at the function level, block level or instruction level. In particular, value profiling of registers (address registers for instance) is possible. However, there seems to be no global analysis of the collected values, necessary for the hot-path computation (Section 4.1) and for the computation of prefetched array sections (Section 4.3). Moreover, Pin does not collect profile information at the loop level and does not correlate information gathered with static information. This implies that identifying a compiler optimization that does not fit the input data (such as deep software pipeline or prefetching for a loop with a small iteration count) is not possible with Pin, whereas MAQAO, thanks to MAQAOPROFILE and its static analysis module, enables this kinds of analysis. EEL (Executable Editing Library) [25] belongs to the same categories of tools. This C++ library allows to edit a binary and add code fragment on edges of dissambled application CFG. Therefore it can be used as a foundation for an analysis tool but does not provide performance analysis by itself. Currently EEL is available on SPARC processors.

HPCview [26] and Finesse [27] (this one being more oriented toward parallelization) address the analysis problem from static and dynamic sides. HPCview tackles the same problem as MAQAO: the complex interaction between source code, assembly, performance and hardware monitors. HPCview presents a well-designed GUI based on web browser, displaying simultaneous views of source, assembly code and dynamic information. This interface is connected to a database storing for each statement of the assembly code a summary of its dynamic information. Based on control flow graph and a tool named bloop, HPCview builds abstracted representation of code loop structures (using an XML interface). Some important differences should be underscored: while a database is embedded in the application, end-user has only limited opportunity to explore the code and define new queries. HPCview also lacks value profiling which can lead to powerful, yet simple to implement optimizations such as code versioning.

Shark [28], [29], developed by Apple offers a comprehensive interface for performance problem. As MAQAO it is located at the assembly level for its analyses, displays source code as well as profiling information. However Shark lacks instrumentation and value profiling, code structures are not displayed and the performance analysis delivers currently a limited number of messages concerning alignment, unrolling or altivec usage (vectorization). Additionally Shark does not offer scripting language or standard database. Nevertheless it remains an advanced interface, with an extensive

support of dynamic behavior (including call stack, garbage collection, binary analysis), and it underlines the need to think performance software beyond gprof.

7. Conclusion

MAQAOPROFILE is an instrumentation module inside a performance evaluation tool, MAQAO. We have shown that the combination of this instrumentation, value-profiling module, with static assembly code analysis delivers precise diagnoses for code tuning that are out of reach of a stand-alone instrumentation approach. Indeed, MAQAO is able to drive the instrumenter and to combine static and dynamic analyses in order to refine the code performance analysis. Moreover, it offers capabilities complementary to the standard performance counter based tools.

MAQAOPROFILE's ability to produce different result level enables user to choose the granularity of the information provided and to define the lag he wants to introduce in his code. Based on MAQAOPROFILE and the performance advisor module, the performance analysis methodology proposed allows to perform efficient code optimization. Currently MAQAO analyzes and instruments codes generated by different compilers (`icc` and `gcc` on IA32 and IA64). The performance analysis is still under development for IA32 architecture (coupling static and dynamic results). Ports for other architectures, in particular for embedded architectures such as ST200 (from STMicro) are planned for future works. In the short term, the GUI will be re-designed with a Web2.0 interface. The idea is to go toward a client/server architecture, where MAQAO would be executed remotely and driven through a web browser.

Finally, we plan to further develop optimization methods based on the performance metrics of MAQAO. These methods would transform the assembly codes according to the performance analysis and profile information achieved. The transformations considered are versioning, unroll and jam, peeling. This approach, in some ways similar to a compiler profile-guided optimization phase, would take advantage of the accuracy of the performance prediction on the assembly code.

References

[1] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva and W. Jalby MAQAO: Modular Assembler Quality Analyzer and Optimizer for Itanium 2 In *Workshop on EPIC architectures and compiler technology*, San Jose, 2005.

[2] S. Eranian, Perfmon project home page: www.hpl.hp.com/research/linux/perfmon HP Labs

[3] J. Dongarra, K. S. London, S. Moore, P. Mucci, D. Terpstra, H. You and Min Zhou. Experiences and Lessons Learned with a Portable Interface to Hardware Performance Counters. *IEEE Int. Conf. Parallel and Distributed Processing Symp.*, 2003: 289

[4] Montecito Processor. http://en.wikipedia.org/wiki/Montecito_(processor)

[5] M. Frigo and S. G. Johnson, The Design and Implementation of FFTW3, *Proc. of the IEEE*, 93 (2), 216-231, 2005, Special Issue on Program Generation, Optimization, and Platform Adaptation,

[6] *Intel IA-64 Architecture Software Developer's Manual, Volume 3: Instruction Set Reference*, revision 2.1 edition. http://developer.intel.com/design/itanium/family.

[7] Intel Itanium2 Processor Reference Manual for Software Development and Optimization, *http://download.intel.com/design/Itanium2/manuals/25111003.pdf*

[8] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva and W. Jalby Exploring Application Performance: a New Tool for a Static/Dynamic Approach. In *Los Alamos Computer Science Institute Symp.*, Santa Fe, NM, 2005.

[9] B. Calder, P. Feller and A. Eustace Value Profiling, *Proc. of ACM IEEE Int. Symp. on Microarchitecture*, December 1-3, 1997, Research Triangle, North Carolina

[10] J. R. Larus, Whole program paths, *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1999, pages 259–269,

[11] G. Pokam and F. Bodin, An Offline Approach for Whole-Program Paths Analysis Using Suffix Arrays, *Proc. of Int. Workshop on Languages and Compilers for Parallel Computing*, 2004, pages 363–378

[12] R. Bellman, On a Routing Problem Quarterly of Applied Mathematics 16(1), pp.87-90, 1958

[13] Standard Performance Evaluation Corporation, *http://www.spec.org/cpu2000/results/res2006q2/*

[14] G. Doshi and R. Krishnaiyer and K. Muthukumar Optimizing software data prefetches with rotating registers *Proc. of Int. Conf. on Parallel Architectures and Compilation Techniques* 2001, Barcelona, Catalunya, Spain

[15] J. Kohn and W. Williams, *J. of Parallel and Distributed Computing* 1993 vol. 18 Issue: 2, p. 205–222

[16] R. Ierusalimschy, L. Henrique de Figueiredo and W. Celes Filho, *Lua — an Extensible Extension Language*, *J. Software Practice and Experience*, 26(6):635–652, june 1996. http://www.lua.org

[17] Intel Corporation. VTune Performance Analyzer http://www.intel.com/software/products/vtune

[18] R. Hundt, HP Caliper: An Architecture for Performance Analysis Tools, *Proc. of Workshop on Industrial Experiences with Systems Software*, October, 2000, San Diego, CA, USA. USENIX 2000 http://www.hp.com/go/caliper

[19] B. Wylie, B. Mohr and F. Wolf Holistic hardware counter performance analysis of parallel programs, *Proc. of Parallel Computing*, Malaga, Spain, 12-16 Sept., 2005.

[20] E. Rohou, F. Bodin, A. Seznec, G. Le Fol, F. Charot and F. Raimbault. SALTO : System for Assembly-Language Transformation and Optimization. RR-2980, 27 p., citeseer.ist.psu.edu/rohou96salto.html

[21] L. De Rose, T. Hoover and J. K. Hollingsworth, The Dynamic Probe Class Library: An Infrastructure for Developing Instrumentation for Performance Tools, www.ptools.org/projects/dpcl *IEEE Int. Conf. Parallel and Distributed Processing Symp.*, 2001: 66

[22] B. R. Buck and J. K. Hollingsworth, An API for runtime code patching *J. of High Performance Computing Application*, 14(4):317-329, 1994.

[23] A. Srivastava and A. Eustace. ATOM - A System for Building Customized Program Analysis Tools. *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation* 1994: 196-205

[24] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation *Proc. of ACM IEEE Int. Symp. on Microarchitecture*, Portland, OR., 2004.

[25] J. R. Larus and E. Schnaar. EEL: Machine-Independent Executable Editing, *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation*, June 1995.

[26] J. Mellor-Crummey, R. Fowler and G. Marin. HPCView: A tool for top-down analysis of node performance. *Computer Science Institute Symp.*, Santa Fe, NM, October 2001.

[27] N. Mukherjee, G.D. Riley and J.R. Gurd. FINESSE: A Prototype Feedback-guided Performance Enhancement System. *Proc. of IEEE Int. Conf. Parallel and Distributed Processing*, 2000, Rhodes, Greece, January 2000.

[28] Optimizing Your Application with Shark 4. http://developer.apple.com/tools/shark_optimize.html

[29] Optimize with Shark: Big Payoff, Small Effort. http://developer.apple.com/tools/sharkoptimize.html

Global Multi-Threaded Instruction Scheduling: Technique and Initial Results

Guilherme Ottoni David I. August

Department of Computer Science
Princeton University
{ottoni, august}@princeton.edu

Recently, the microprocessor industry has reached hard physical and micro-architectural limits that have prevented the continuous clock-rate increase, which had been the major source of performance gains for decades. These impediments, in conjunction with the still increasing transistor counts per chip, have driven all major microprocessor manufacturers toward Chip Multiprocessor (CMP) designs. Although CMPs are able to concurrently pursue multiple threads of execution, they do not directly improve the performance of most applications, which are written in sequential languages. In effect, the move to CMPs has shifted even more the task of improving performance from the hardware to the software. In order to support more effective parallelization of sequential applications, computer architects have proposed CMPs with light-weight communication mechanisms [26, 24, 22]. Despite such support, proposed multi-threaded scheduling techniques have generally demonstrated little effectiveness [15, 16] in extracting parallelism from general-purpose, sequential applications. We call these techniques *local multi-threaded scheduling*, because they basically exploit parallelism within straight-line regions of code. A key observation of this paper is that local multi-threaded techniques do not exploit the main feature of CMPs: the ability to concurrently execute instructions from different control-flow regions. In order to benefit from this powerful CMP characteristic, it is necessary to perform *global multi-threaded scheduling*, which simultaneously schedules instructions from different basic blocks to enable their concurrent execution. This paper presents algorithms to perform global scheduling for communication-exposed multi-threaded architectures. By *global* we mean that our technique simultaneously schedules instructions from an arbitrary code region. Very encouraging preliminary results, targeting a dual-core Itanium 2 model, are presented for selected benchmark applications.

1 Introduction

In the last few years, the microprocessor industry has been undergoing what is being considered one of its major changes. Suddenly, hard physical limitations, aligned with the diminishing returns of micro-architectural improvements, have prevented the design of faster microprocessors. Nevertheless, the number of transistors available on a chip continues to increase exponentially over time. Combined, these factors have directed all major microprocessor manufacturers toward multi-core designs, also known as chip multiprocessors (CMPs). Unfortunately, while CMPs increase throughput for multiprogrammed and multi-threaded codes, many important applications are single-threaded and thus do not benefit from CMPs.

This change in paradigm has resulted in a tremendous interest on parallel applications. Although ideally programmers could rewrite all the applications in a parallel paradigm, parallel programming has long been recognized as more time-consuming, error-prone, and harder to debug than its sequential counterpart. Furthermore, it is impractical to rewrite all the existing applications. A more viable alternative is to use parallelizing compilers to automatically generate parallel code from sequential programs. Unfortunately, despite decades of research on parallelizing compilers, these have only proved effective in the restricted domain of scientific applications, which have remarkably regular array-based memory accesses and little control flow.

Because the parallelism available in non-scientific applications is typically much more fine-grained, computer architects have proposed simple hardware support mechanisms to enable light-weight fine-grained communication [26, 24, 22, 21], generally called *scalar operand networks*. These mechanisms typically consist of an on-chip interconnect between the processor cores, and special `produce` and `consume` instructions to communicate scalar values from one core to another. To the software, these communication mechanisms look like sets of hardware-implemented queues. Extracting parallelism for these processors consists of partitioning the computation into threads, to execute on different cores, and inserting communication instructions to satisfy inter-thread dependences. The parallelism exposed by these processors is of finer granularity than what is typically exploited by programmers in parallel systems, making it even harder to manually explore these opportunities. Therefore, generating code that exploits these opportunities is better performed by a compiler's instruction scheduler.

Instruction scheduling techniques can be classified as

either *local* or *global*. While local techniques schedule the instructions of each basic block independently, global approaches simultaneously consider instructions from different basic blocks. Most of the existing multi-threaded scheduling techniques are based on local scheduling [15, 16]. We call these techniques *local multi-threaded* (LMT) scheduling.

One of our key observations is that LMT scheduling techniques do not exploit the main advantage brought by multi-threaded architectures: the ability to simultaneously follow different execution paths in different processor cores. Given the typically small size of basic blocks in general-purpose applications, we believe it is crucial to exploit parallelism beyond basic block boundaries in order to successfully extract parallelism from these applications. As an example, consider the sample C code in Figure 1. Although these loops may iterate for a large number of iterations, very little instruction-level parallelism is available within each individual basic block. For such control-intensive codes, any local scheduling technique will hardly extract any thread-level parallelism. Notice, however, that the computation in each loop is independent, and therefore they can be executed in parallel. Nevertheless, in order to exploit such sources of parallelism, it is necessary to perform *global multi-threaded* (GMT) scheduling, i.e. to simultaneously consider instructions from different basic blocks during scheduling. The major complication of going from any local analysis or optimization to its corresponding global version is the presence of control flow. In this paper, we demonstrate how control flow can effectively be handled to enable GMT scheduling. Our technique combines a global multi-threaded list scheduling heuristic with a novel global multi-threaded code generation algorithm. These algorithms are based on a *Program Dependence Graph* (PDG) representation [6], which includes both data and control dependences.

Overall, this paper makes the following contributions:

1. It introduces the concept of global multi-threaded (GMT) scheduling, which we believe is key to fully take advantage of multi-threaded architectures, in particular to parallelize general-purpose applications.
2. It shows how to handle control flow in order to enable GMT scheduling, and presents a novel global multi-threaded list scheduling heuristic.
3. It presents an effective dynamic programming algorithm to efficiently perform GMT scheduling on large code regions composed of complex loop nests.
4. It presents a general algorithm to generate multi-threaded code from arbitrary partitions of the instructions among the threads. This algorithm is a generalization for arbitrary CFGs of the algorithm used for loop scheduling in [19], and it can be used with *any* GMT scheduling heuristic.

```
s1 = 0;
s2 = 0;
for (p = head; p != NULL; p = p->next) {
  s1 += p->value;
}
for (i = 0; a[i] != 0; i++){
  s2 += a[i];
}
printf("%d\n", s1 * s1 / s2);
```

Figure 1. Example code in C.

5. It shows initial promising experimental results targeting a highly accurate dual-core Itanium 2 model.

The rest of the paper is organized as follows. Section 2 gives some background on PDGs. We present our GMT scheduling heuristics in Section 3, followed by our multi-threaded code generation algorithm in Section 4. In Section 5, we present experimental results. Finally, we discuss related work in Section 6, and conclude in Section 7.

2 Program Dependence Graphs

Local scheduling techniques operate by constructing a data dependence graph representing all data dependences that must be respected in order to keep the original program's semantics. At a low-level representation, data dependences can take two forms: register data dependences, or memory data dependences. Furthermore, data dependences can be of three kinds, depending on whether the involved instructions read or write the data location [13]: *flow dependence*, which goes from a write to a read; *anti-dependence*, which goes from a read to a write; and *output dependence*, which goes from a write to another write. Register data dependences can be efficiently and precisely computed through data-flow analysis. For memory data dependences, compilers typically rely on the result of pointer analysis to determine which loads and stores may access the same memory locations. Although computationally much more complicated, practical existing pointer analysis can typically disambiguate a large number of non-conflicting memory accesses even for type-unsafe languages like C.

The key addition from a local scheduling to a global scheduling technique is the necessity of handling control flow. In other words, in addition to the data dependences typically used for local scheduling, it is necessary to add *control dependence* arcs to the dependence graph. A control dependence arc from a branch instruction X to an instruction Y means that, depending on the direction taken at X, Y either must or may not be executed. Dependence graphs including both data and control dependences are generally called *Program Dependence Graphs* (PDGs) [6]. Cytron et al. [4] proposed an efficient algorithm to compute control dependences for arbitrary CFGs, based on *post-dominance frontiers*.

```
(A)  B1: move   r1 = 0         ;; r1 contains s1
(B)      move   r2 = 0         ;; r2 contains s2
(C)      load   r3 = [head]    ;; r3 contains p
(D)  B2: branch r3 == 0, B4
(E)  B3: load   r4 = [r3]      ;; load p->value
(F)      add    r1 = r1, r4
(G)      load   r3 = [r3+4]    ;; load p->next
(H)      jump   B2
(I)  B4: move   r5 = @a        ;; r5 pts. to a[i]
(J)  B5: load   r6 = [r5]      ;; load a[i]
(K)      branch r6 == 0, B7
(L)  B6: add    r2 = r2, r6
(M)      add    r5 = r5, 4
(N)      jump   B5
(O)  B7: mult   r7 = r1, r1
(P)      div    r8 = r7, r2
```

Figure 2. Low-level IR for the code in Figure 1.

2.1 PDG for Global Multi-Threaded Scheduling

The PDG for global multi-threaded (GMT) scheduling contains one vertex for each instruction in the code, with the exception of `jump` instructions. These instructions are left aside because their effect is embedded in the control dependences. Besides this, they only serve to put the basic blocks of the CFG into a linear representation.

We now precisely define the PDG dependence arcs necessary for our GMT scheduling. We denote V_G and E_G, respectively, the sets of vertices and arcs of a graph G.

- Register data dependences: only flow dependences through registers need to be considered, and anti- and output dependences can be ignored. The reason for this is that, if two instructions involved in an anti- or output dependence are scheduled on different threads, they will execute in two processors with different register sets. In other words, the use of different register sets automatically eliminates false dependences. Additionally, for instructions scheduled on the same thread, our algorithm preserves their order, thus naturally respecting intra-thread dependences. A register dependence from instruction X to Y involving r_i is denoted $X \rightarrow^{r_i} Y$.
- Memory data dependences: for memory, all flow, anti- and output dependences need to be taken into account, as the memory store is shared by the threads. Memory dependences are denoted $X \rightarrow^{M} Y$.
- Direct control dependences: our PDG includes control dependences. We denote $X \rightarrow^{T} Y$ for taken and $X \rightarrow^{F} Y$ for not-taken branch directions.
- Transitive control dependences: for each dependence $(X \rightarrow Y) \in E_{PDG}$, and for each branch instruction B on which X is dependent, a transitive control dependence arc $B \rightarrow^{*} Y$ is added to E_{PDG}. The reason for these dependences will become evident in Section 4, when we describe our multi-threaded code generation algorithm.

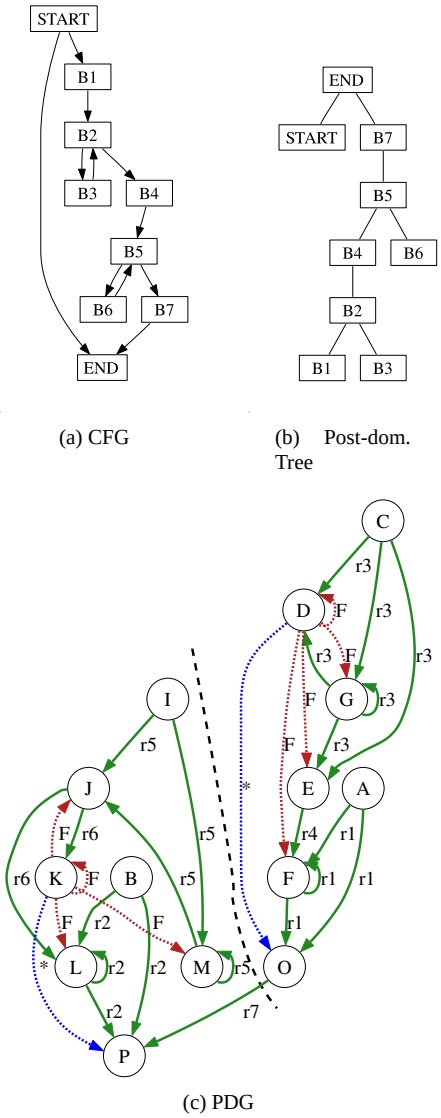


Figure 3. (a) CFG, (b) post-dominance tree, and (c) PDG.

Figure 2 illustrates a low-level representation for the code in Figure 1, and Figure 3 contains the corresponding CFG, post-dominance tree, and PDG.

Using the PDG constructed as described above, different GMT scheduling heuristics can be applied to choose a global schedule, i.e. a partition of the instructions among the threads. For example, for the PDG in the example of Figure 3(c), a heuristic may decide to partition the code in two threads as depicted by the dashed line. This partition corresponds to scheduling each loop of Figure 1 onto a separate thread.

3 Global Multi-Threaded Instruction Scheduling

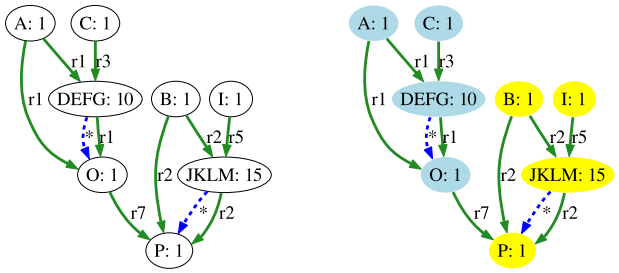
In this section, we describe in details our GMT scheduling algorithms, and illustrate them on the example of Figure 1. This section is only concerned with the thread-level scheduling decisions, i.e. mapping the instructions of the original sequential program onto threads. The multi-threaded code generation algorithm is presented in Section 4. Because each thread generated by our technique is intended to execute on a different core, we interchangeably say that an instruction is scheduled on a thread or core.

Although a multi-threaded instruction scheduler can be combined with a traditional single-threaded scheduler, we opted not to do so in this work. One reason for this is that the multi-threaded code generated by our technique can be further optimized before the actual assembly code generation. Additionally, exposing all the machine details to the GMT scheduler would make its implementation more complex. Instead, we preferred to keep our GMT scheduler simpler by providing it with just a few key characteristics of the target processor, namely the number of cores and the issue-width of each core. A latency of one cycle is assumed for most instructions (except for function calls), and no information about structural hazards is used.

Our GMT scheduler uses a PDG as intermediate representation for both scheduling decisions and code generation. Because our technique is global, targeting arbitrary code regions, it must deal with the possibility of cycles in a PDG. Scheduling of cyclic graphs is more complex than scheduling of acyclic graphs. The goal of a scheduler is to minimize the critical (i.e. longest) path through the graph. Although scheduling of acyclic graphs in the presence of resource constraints is NP-hard, at least finding the critical path in such graphs can be solved in time linear, through a topological sort. For cyclic graphs, however, even finding the longest path is NP-hard [8].

Given the inherent difficulty of the global scheduling problem for cyclic code regions, we use a simplifying approach that reduces it to the acyclic scheduling problem, for which well-known heuristics based on list scheduling [9] exist. In order to reduce the cyclic scheduling problem to an acyclic one, we make two simplifications to the problem. First, when scheduling a given code region, each of its inner loops is coalesced to a single node, with an aggregated latency that assumes its average number of iterations. Secondly, if the code region being scheduled is a loop, all the loop-carried dependences are disregarded. To deal with the possibility of irreducible code, we use a loop hierarchy that includes irreducible loops [10]. It is important to notice that these simplifying assumptions are used for scheduling decisions only; our code generation algorithm takes all dependences into account to generate correct code.

To distinguish from a full PDG, we call the dependence



(a) HPDG

(b) Clustered HPDG

	Core 0		Core 1	
cycle	issue 0	issue 1	issue 0	issue 1
0	A	C	B	I
1	D E	F G	J K	L M
2	D E	F G	J K	L M
3	D E	F G	J K	L M
4	D E	F G	J K	L M
5	D E	F G	J K	L M
6	D E	F G	J K	L M
7	D E	F G	J K	L M
8	D E	F G	J K	L M
9	D E	F G	J K	L M
10	D E	F G	J K	L M
11	O		J K	L M
12	prod r7		J K	L M
13			J K	L M
14			J K	L M
15			J K	L M
16			cons r7	
17			P	

(c) Schedule

Figure 4. (a) HPDG for the PDG from Figure 3(c). (b) Clustered HPDG. (c) Chosen Schedule.

graph for a region with its inner loops coalesced and its loop-carried dependences ignored a *Hierarchical Program Dependence Graph* (HPDG). In a HPDG, the nodes represent either a single instruction, called a *simple node*, or a coalesced inner loop, called a *loop node*. Figure 4(a) illustrates the HPDG corresponding to the PDG from Figure 3(c). The nodes are labeled by their corresponding nodes in the PDG, followed by their estimated latency. There are only two loop nodes in this example: DEFG and JKLM.

Even after eliminating the cycles in the PDG, control flow still poses additional complications to GMT instruction scheduling that do not exist for local scheduling. In local scheduling, there is a guarantee that all instructions will execute, i.e. all instructions being scheduled are *control equivalent*. Therefore, as long as the dependences are satisfied and resources are available, the instructions can safely be issued simultaneously. The presence of arbitrary control

flow complicates the matters for GMT scheduling. First, control flow causes many dependences not to occur during the execution. Second, not all instructions being scheduled are control equivalent anymore. For example, the fact that instruction A executes may not be related to the execution of B, or may even imply that B will not execute. To deal with the different possibilities, we introduce three different *control relations* among instructions, which are used during our scheduling algorithms.

Definition 1 (Control Relations) *Given two HPDG nodes A and B, we call them:*

1. Control Equivalent, *if both A and B are simple nodes with the same direct control dependences.*
2. Mutually Control Exclusive, *if the execution of A implies that B does not execute, or vice-versa.*
3. Control Conflicting, *otherwise.*

To illustrate these relations, consider the HPDG from Figure 4(a). In this example, A, B, C, I, O and P are all control equivalent. Nodes DEFG and JKLM are control conflicting with every other node. No pair of nodes is mutually control exclusive in this example.

Another problem intrinsic to GMT scheduling is that the multiple threads generated will execute on different cores, and so it is necessary to take the communication overhead into account while making scheduling decisions. For example, even though two instructions can be issued in parallel on different threads, this might not be profitable due to necessary communication to move their operands from one core to another. To address this problem, we use a *clustering* pre-scheduling pass on the HPDG, which takes into account the inter-core communication overhead. The goal of this pass is to cluster together HPDG nodes that are likely to not benefit from schedules that assign them to different threads. Section 3.1 explains the clustering algorithm we use, and our multi-threaded instruction scheduling heuristic is described in Section 3.2.

3.1 Clustering Algorithm

Our clustering algorithm is an adaptation of the *Dominant Sequence Clustering (DSC)* algorithm [29], widely used for task scheduling in parallel computing. Here we briefly describe DSC, and point out the modifications we incorporated to more adequately deal with powerful ILP processor cores.

The DSC algorithm, like all multi-processor task scheduling algorithms, performs clustering on a directed acyclic graph (DAG). Therefore, because of the simplifications we performed to reduce our cyclic scheduling problem into an acyclic one, we can rely on previous research on multi-processor task scheduling. We chose to use DSC because it has been shown to be both effective and efficient, being able to handle graphs with thousands of nodes [29]. Efficiency is important for instruction scheduling because of the potentially huge number of nodes in a HPDG.

The DSC algorithm assumes that each cluster will be executed on a different processor (core for us). The later scheduling pass may schedule multiple clusters on the same thread to cope with a smaller number of processors.

DSC starts by assigning each instruction to a different cluster. The critical path passing through each node of the graph is then computed, considering both the execution latencies of nodes and the communication latencies. The communication latency is assumed to be zero if and only if the nodes are in the same cluster. DSC then processes each node at a time, following a topological order prioritized by the nodes' critical path length. At each step, the benefit of merging the node being processed with each of its predecessors is analyzed. The advantage of merging a node with another cluster is that the communication latency from nodes in that cluster will be saved. The downside of merging is that the instructions assigned to the same cluster are assumed to execute sequentially, in the order they are added to the cluster. Therefore, the delayed execution after a merge may outweigh the benefits of the saved communication. The node being processed is then merged with its predecessors' cluster that reduces this node's critical path the most. If all such cluster increase the critical path, this node is left alone in its own cluster.

In this work, we use a slight modification of the DSC algorithm to deal with the ILP power available in modern processor cores. To do that, we do not assume a sequential execution inside each cluster. Instead, we assume the node being merged will be issued at the earliest cycle such that: (a) its inter-cluster dependences are satisfied (including the communication cost), (b) its intra-cluster dependences are fulfilled, (c) its *control conflicting* nodes inside the cluster are finished, and (d) there is an issue slot available.

In addition, in our DSC variation, we use a more refined breakdown of the communication overhead components, which suits better the inter-core communication mechanism we assume. Whenever there is an inter-cluster dependence from instruction A to instruction B, we assume the following communication latencies:

- *Producing Latency* in A's cluster, after A executes.
- *Consuming Latency* in B's cluster, before B executes.
- *Communication Latency*, which is added to A's finish cycle to estimate when A's value will be available for use by B.

Finally, in our DSC implementation, we perform a post-pass to eliminate some trivial clusters. Specifically, we look for all clusters that contain a single simple node and that has dependences with instructions from a single cluster. When

such a trivial cluster is found, it is merged with its single adjacent cluster.

Figure 4(b) illustrates the clusters resulting from this algorithm. For simplicity, we assume here that the producing, consuming, and communication latencies are all one cycle. Initially, each node is its own cluster. The first nodes to be processed are B and L, which have priority (i.e. the length of the longest, critical path through it) equal to 23. For example, for B, the longest path includes 17 cycles of execution latency, plus 2 cycles of producing latency (for B and JKLM), 2 cycles of communication latency (for arcs B→JKLM and JKLM→P), and 2 cycles of consuming latency (for both JKLM and P). As neither B nor I have predecessors, they are left on their own clusters. Next, node JKLM is processed, which also has priority 23. Merging this node with any of its predecessors will not increase its priority, so we arbitrarily merge it with B. After that, nodes A and C are processed, and each remains in its own cluster. Notice that, even though node P has higher priority, it does not have all its predecessors processed yet. Next, node DEFG is processed and, similarly to what happened to JKLM, it is arbitrarily merged with one of its predecessors, A. At this point, node O is processed, and it is merged with its only predecessor cluster, which contains A and DEFG. Finally, P is processed, and it is merged with the cluster containing B and JKLM, what reduces the critical path to 20. At this point, the trivial-cluster elimination post-pass is performed, and two trivial clusters are merged: (1) node C is merged with the cluster containing A, DEFG, and O; and (2) node I is merged with the cluster containing B, JKLM, and P.

3.2 Global Multi-Threaded List Scheduling

After the clustering pass on the HPDG, the actual scheduling decisions are made. Here again, because of our reduction to an acyclic scheduling problem, we can rely on well-known acyclic scheduling algorithms. In particular, we use a form of list scheduling with resource constraints, with some adaptations to better deal with our problem. This section describes list scheduling and our enhancements to it.

The basic list scheduling algorithm assigns priorities to nodes and schedules each node following a prioritized topological order. Typically, the priority of a node is computed as the longest path from it to a leaf node. A node is scheduled at the earliest time that satisfies its input dependences and that conforms to the currently available resources.

For traditional, single-threaded instruction scheduling, the resources correspond to the processor's functional units. For GMT instruction scheduling, there are two levels of resources: the target processor contains multiple cores, and each core has a set of functional units. Considering these two levels of resources, instead of simply assuming the total number of functional units in all cores, is important for many reasons. First, it enables us to consider the communication overhead to satisfy dependences between instructions scheduled on different cores. Furthermore, it allows us to benefit from opportunities available in a *global* scheduling problem, in particular the simultaneous issue of control conflicting instructions. Because each core has its own control unit, control-conflicting instructions can be issued in different cores in the same cycle.

Thread-level scheduling decisions are made when scheduling the first node in a cluster. At this point, the best thread is chosen for that particular cluster, given what has already been scheduled. When scheduling the remaining nodes of a cluster, we simply schedule it on the thread previously chosen for this cluster.

The choice of the best thread to schedule a particular cluster takes into account a number of factors. Broadly speaking, these factors try to find a good equilibrium between two conflicting goals: maximizing the parallelism, and minimizing the inter-thread communication. For each thread, we compute the total overhead of assigning the current cluster to it. This total overhead is the sum of the following components:

1. *Startup Overhead*: this is the difference between the first cycle in which the node in consideration can be scheduled on the given thread and the current cycle.
2. *Communication Overhead*: this is the total number of cycles that will be necessary to execute all `produce` and `consume` instructions to satisfy dependences between this cluster and instructions in clusters already scheduled on different threads.
3. *Resource-Conflict Overhead*: this is an estimated number of cycles by which the execution of this cluster will be delayed when executing in this thread, considering the current load of unfinished instructions already assigned to this thread. This takes into account both the average resource utilization per cycle for the unfinished instructions, as well as the total latency of the current cluster. In effect, this overhead is more important for larger clusters, and it is useful to improve the load balance among threads.
4. *Control-Conflict Overhead*: this is an estimated number of cycles in which instructions in this cluster will not be able to execute in this thread due to control conflicts with unfinished instructions of other clusters already scheduled on this thread. To compute this estimate, we use the latency of the unfinished instructions in other clusters assigned to this thread, weighted by the probability that a control conflict will impede the issue of each instruction in the cluster being scheduled. This control conflict probability is computed as the latency of the unfinished instructions that are con-

trol conflicting with the one being considered, over the total latency of all unfinished instructions.

Once we choose the thread in which a HPDG node is to be scheduled, it is necessary to estimate the cycle in which that node can be issued in this core. Although we do not perform the actual scheduling at this point, this estimate is used to guide the GMT scheduling for the remaining nodes.

In order to find the estimated cycle in which a node can be issued in the chosen thread, it is necessary to consider two restrictions. First, we need to make sure that the node's input dependences will be satisfied at the chosen cycle. For inter-thread dependences, it is necessary to account for the communication latency and corresponding `consume` instructions overhead. Second, the chosen cycle must be such that there are available resources in the chosen core, given the other nodes already scheduled on it. However, not all the nodes already scheduled on this thread should be considered. Resources used by nodes that are mutually control exclusive to this one are considered available, as these nodes will never be issued simultaneously. On the other hand, the resource utilization of control equivalent nodes must be taken into account. Finally, the node cannot be issued in the same cycle as any previously scheduled node that has a control conflict with it. This is because each core has a single control unit, but control-conflicting nodes have unrelated conditions of execution. Notice that, however, for target cores that support predicated execution, this is not necessarily valid: two instructions with different execution conditions may be issued in parallel. But even for cores with predication support, loop nodes cannot be issued with anything else.

We now show how our list scheduling algorithm works on our running example. For illustration purposes, we use as target a dual-core processor that can issue two instructions per cycle in each core (see Figure 4(c)). The list scheduling algorithm processes the nodes in the clustered HPDG (Figure 4(b)) in topological order. The nodes with highest priority (i.e. longest path to a leaf) are B and I. B is scheduled first, and it is arbitrarily assigned to core 1's first slot. Next, node I is considered and, because it belongs to the same cluster as B, the core of choice is 1. Because there is available resource (issue slot) in core 1 at cycle 0, and the fact that B and I are control equivalent, I is scheduled on core 1's issue slot 1. At this point, we may schedule nodes A, C, or JKLM. Even though JKLM has the highest priority, its input dependences are not satisfied in the cycle being scheduled, cycle 0. Therefore, JKLM is not a *candidate* node in the current cycle. So node A is scheduled next, and the overheads described above are computed for scheduling A in each thread. Even though thread 1 (at core 1) has lower communication overhead (zero), it has higher startup, control-conflict, and resource-conflict overheads. Therefore, core 0 is chosen for node A. The algorithm then proceeds, and the remaining scheduling decisions are all cluster-based. Figure 4(c) illustrates the final schedule built and the partitioning of the instructions among the threads.

3.3 Handling Loop Nests

Although our scheduling algorithm follows the clusters formed a priori, we make an exception when handling inner loops. The motivation to do so is that inner loops may fall on the region's critical path, and they may also benefit from execution on multiple threads.

We handle inner loops as follows. For now, assume that we have an estimate for the latency to execute one invocation of an inner loop L_j using a number of cores i from 1 up to the number N of cores on the target processor. Let $latency_{L_j}(i)$, $1 \leq i \leq N$, denote these latencies. Considering L_j's control conflicts, we compute the cycle in which each core will finish executing L_j's control-conflicting nodes already scheduled on it. From that, we can compute the earliest cycle in which a given number of cores i will be available for L_j, denoted by $cycle\_available_{L_j}(i)$, $1 \leq i \leq N$. With that, we choose to schedule this loop node on a number of cores k such that $cycle\_available_{L_j}(k) + latency_{L_j}(k)$ is minimized. Intuitively, this will find the best balance between the wait to have more cores available and the benefit from executing the loop node on more threads. If more than k threads are available at $cycle\_available(k)$ (e.g. all threads will be available in the same cycle, but we do not need all of them), then we pick the k threads among them with which the loop node has more affinity. The affinity is computed as the number of dependences between this loop node and nodes already scheduled on each thread.

The question that remains now is: how do we compute the $latency_{L_j}(i)$ for each child loop L_j in the HPDG? Intuitively, this is a recursive question, since what we have been doing is scheduling a code region on multiple threads, with the goal of minimizing its execution latency. This naturally leads to a recursive solution. But even better, we can apply dynamic programming to efficiently solve this problem in polynomial time. In addition, were our list scheduling algorithm perfect, this would be able to compute the *optimal* scheduling for an arbitrary code region.

More specifically, our dynamic programming solution works as follows. First, we compute the loop hierarchy for the region we want to schedule. This can be viewed as a loop tree, where the root represents the whole region (in case the region is not a loop itself). The algorithm then proceeds bottom-up on this loop tree and, for each tree node L_j (either a loop or the whole region) it applies the GMT list scheduling algorithm to compute the latency to execute one iteration of that loop, with a number of threads i varying from 1 to N. This latency returned by the list

scheduling algorithm is then multiplied by the average number of iterations per invocation of this loop, resulting in the $latency_{L_j}(i)$ values to be used for this loop node when scheduling its parent. In the end, we choose the best schedule for the whole region by picking the number of threads k for the loop tree's root, R, such that $latency_R(k)$ is the minimum. The corresponding partitioning of instructions onto threads can be obtained by keeping and propagating the partition $partition_{L_j}(i)$ of instructions corresponding to the value of $latency_{L_j}(i)$.

As a final note, we point that this dynamic programming approach can be used in a general framework that considers other loop parallelization and scheduling techniques, such as DOALL, DOACROSS and DSWP [19], besides the GMT list scheduling described here. The evaluation of such general framework is beyond the scope of this paper.

4 Multi-Threaded Code Generation

We now describe our Multi-Threaded Code Generation (MT-CG) algorithm. For any global schedule chosen, this algorithm generates corresponding multi-threaded code, automatically inserting the communication and synchronization instructions necessary to preserve the program's dependences.

Figure 5 presents the MT-CG algorithm, which takes as input the original control-flow graph (*CFG*), the *PDG* constructed as described in Section 2.1, and the chosen global schedule (*GS*). As output, this algorithm produces a new CFG for each of the resulting threads, containing its corresponding instructions, and including the necessary communication and synchronization instructions.

In essence, the MT-CG algorithm works as follows. For each of the threads specified by the global schedule, it generates a new CFG with only the necessary basic blocks for this thread. Then, the instructions are inserted in the thread to which they were scheduled. After that, the necessary inter-thread communication and synchronization instructions are inserted into the code. Finally, branch and jump instructions are adjusted to account for missing basic blocks in the new CFGs.

Before going into the details of the algorithm in Figure 5, let us introduce the notation used. P_i denotes a partition (thread) in *GS*, and CFG_i denotes its corresponding control-flow graph. For a given instruction I, $bb(I)$ is the basic block containing I in the CFG, and $point_j(I)$ is the point in CFG_j corresponding to the location of I in the CFG.

The first step of the algorithm is to find the set of the *relevant basic blocks* for each partition (thread) P_i in *GS*. The set of relevant basic blocks for P_i contains the set of blocks that will compose CFG_i. Additionally, CFG_i is carefully constructed so that each of its basic blocks has exactly the same condition of execution as its corresponding block in CFG. The procedure Relevant_BBs, lines 26-31 in Figure 5, describes how to compute such set of basic blocks. This set contains one block for each block in the original CFG that contains either (a) an instruction scheduled to P_i, or (b) an instruction on which any of P_i's instructions depends (i.e. a source of a dependence with an instruction in P_i as the target). The reason for including basic blocks containing instructions in P_i is obvious, as they will hold these instructions in the generated code. The reason for adding the basic blocks containing instructions on which P_i's instructions depend is related to a property used to preserve the semantics in the transformed code: inter-thread communication instructions are inserted at the point of the *source instruction*, so as to keep the exact condition under which this dependence happens. Notice that this particular choice of where to communicate a dependence is somewhat arbitrary, and may not be optimal. However, this choice does simplify the proof of correctness of the algorithm. This choice of where dependences are communicated is also the motive for making the transitive control dependence arcs explicit in the PDG: if these dependence arcs connect instructions scheduled to different threads, these branches need to be communicated so that the inter-thread dependences keep their condition of execution. The call to *create_corresp_bb*$_i(B)$ (line 28) creates the block B_i corresponding to B in CFG_i. The mappings between B and B_i are denoted by: *corresp_bb*$_i(B) = B_i$, and *orig_bb*$(B_i) = B$.

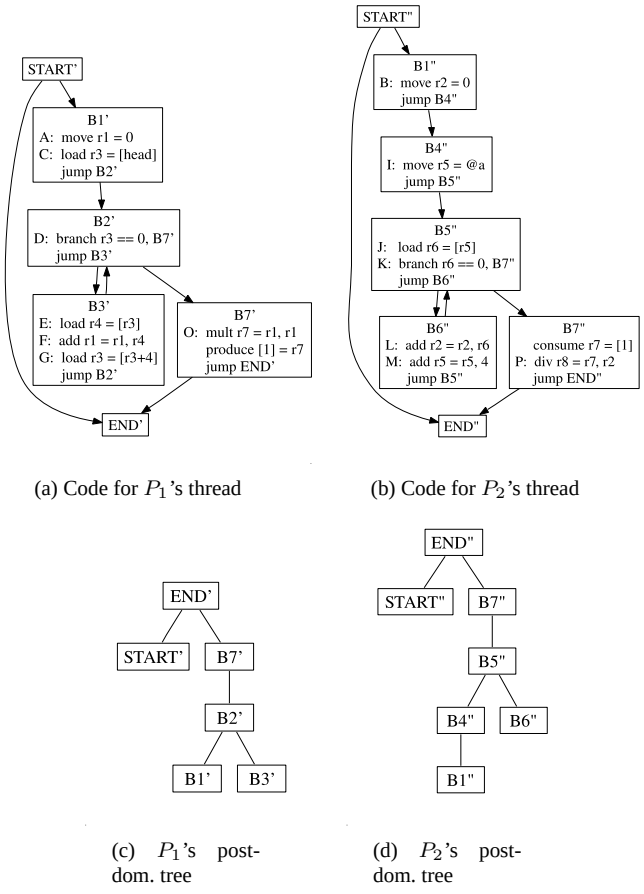
The next step of the MT_CG algorithm (lines 3-5) is to insert the instructions in P_i into their corresponding basic blocks in CFG_i. The instructions are inserted in the same relative order as in the original code, so that intra-thread dependences are naturally satisfied. After this, the code in lines 6-18 inserts communication and synchronization instructions in order to preserve the inter-thread dependences. For each such dependence, a separate communication queue is used[1]. Notice that, as mentioned above, the communication instructions are always inserted at the point corresponding to the instruction that is the source of the dependence. The actual communication instructions inserted depend on the type of the dependence. Register dependences are implemented by communicating the register in question right after the point that it is produced in the source thread. For memory dependences, purely synchronization instructions are inserted to enforce that their relative order of execution is preserved. Finally, control dependences are more involving. In the source thread, before the branch is executed, its register operand is sent. In the thread that is the sink of the dependence, a `consume` instruction is inserted to get the corresponding register value, and then an equivalent branch instruction is inserted to mimic the same control behavior.

[1] A separate queue is used just for simplicity. Later, a queue-allocation algorithm can reduce the number of queues necessary.

```
MT_CG (CFG, PDG, GS)
(1)   for each P_i ∈ GS do
(2)     V_CFG_i ← Relevant_BBs(CFG, PDG, P_i)
(3)   for each I ∈ V_PDG, in original program order, do
(4)     let i be such that I ∈ P_i
(5)     add_last(corresp_bb_i(bb(I)), I)
(6)   for each (I → J) ∈ E_PDG, where I ∈ P_i, J ∈ P_j, P_i ≠ P_j do
(7)     q ← get_free_queue()
(8)     if dep_type(I → J) = r_k then
(9)       add_after(corresp_bb_i(bb(I)), I, "produce [q] = r_k")
(10)      add_before(corresp_bb_j(bb(I)), point_j(I), "consume r_k = [q]")
(11)    else if dep_type(I → J) = M then
(12)      add_after(corresp_bb_i(bb(I)), I, "produce [q]")
(13)      add_before(corresp_bb_j(bb(I)), point_j(I), "consume [q]")
(14)    else   // control dependence
(15)      r_k ← register_argument(I)
(16)      add_before(corresp_bb_i(bb(I)), I, "produce [q] = r_k")
(17)      add_before(corresp_bb_j(bb(I)), point_j(I), "consume r_k = [q]")
(18)      add_before(corresp_bb_j(bb(I)), point_j(I), I)
(19)  for each P_i ∈ GS do
(20)    add START and END nodes to CFG_i
(21)    for each branch I ∈ P_i do
(22)      redirect_target(I, closest_relevant_postdom_i(target(I)))
(23)    for each B ∈ V_CFG_i do
(24)      CRS ← closest_relevant_postdom_i(succ(orig_bb(B)))
(25)      add_last(B, "jump CRS")

Relevant_BBs(CFG, PDG, P_i)
(26)  RB ← ∅
(27)  for each I ∈ P_i do
(28)    RB ← RB ∪ {create_corresp_bb_i(bb(I))}
(29)    for each J | (J → I) ∈ E_PDG do
(30)      RB ← RB ∪ {bb(J)}
(31)  return RB
```

Figure 5. Multi-threaded code generation algorithm.



(a) Code for P_1's thread

(b) Code for P_2's thread

(c) P_1's post-dom. tree

(d) P_2's post-dom. tree

Figure 6. Resulting multi-threaded code and corresponding post-dominance trees.

The last step of the algorithm (lines 19-25) is to insert *START* and *END* nodes in the new CFGs, and to fix the branch targets and insert `jump` instructions to properly connect the basic blocks in each of the new CFGs. Because not all the basic blocks in the original CFG have a corresponding one in each new CFG, finding the adequate branch/jump targets is non-trivial. In order to preserve the control dependences, the branch/jump targets need to be retargeted to the *closest post-dominator basic block B* of the original target/successor, in the original CFG, such that *B* is *relevant* to the new CFG. We call such block *B* the *closest relevant post-dominator*, in the new CFG, of the original target/successor. Notice that such post-dominator basic block always exists as every vertex is post-dominated by *END*, which is relevant to every CFG.

A simple optimization, not illustrated in the algorithm in Figure 5, is that dependences between the same pair of threads that have the same source instruction need only to be communicated once. Moreover, many jump instructions inserted to connect the blocks in each new CFG can be eliminated by code layout and jump optimizations.

We have proved that our MT_CG algorithm preserves all the dependences in the PDG. Combined with Sarkar's result showing that any transformation that preserves all dependences in the PDG also preserves the program's semantics [25], this leads to the correctness proof of the MT_CG algorithm. In interest of space, we omit these proofs.

Figures 6(a)-(b) illustrate the generated code for the two threads corresponding to the global schedule depicted in Figure 3(c). In Figures 6(c)-(d), the post-dominator trees for the new CFGs are illustrated. As can be easily checked, each of the resulting threads contains only its relevant basic blocks, the instructions scheduled to it, the instructions inserted to satisfy the inter-thread dependences, and jumps inserted to connect the CFG. In this example, there is a single pair of `produce` and `consume` instructions, corresponding to the only cross-thread dependence in the schedule shown in Figure 3(c).

By analyzing the resulting code in Figures 6(a)-(b), it should be clear that the resulting threads are able to concurrently execute instructions in different basic blocks of the original code, effectively following different control-flow

Core	Functional Units: 6 issue, 6 ALU, 4 memory, 2 FP, 3 branch L1I Cache: 1 cycle, 16 KB, 4-way, 64B lines L1D Cache: 1 cycle, 16 KB, 4-way, 64B lines, write-through L2 Cache: 5,7,9 cycles, 256KB, 8-way, 128B lines, write-back Maximum Outstanding Loads: 16
Shared L3 Cache	> 12 cycles, 1.5 MB, 12-way, 128B lines, write-back
Main Memory	Latency: 141 cycles
Coherence	Snoop-based, write-invalidate protocol
L3 Bus	16-byte, 1-cycle, 3-stage pipelined, split-transaction bus with round robin arbitration

Table 1. Machine details.

paths. The potential of exploiting such parallelization opportunities is unique to a *global* multi-threaded scheduling, and constitutes its key advantage over *local* multi-threaded scheduling approaches.

5 Evaluation

We implemented our GMT scheduling technique in the Velocity compiler, a research compiler derived from UIUC's IMPACT compiler [1] that targets Itanium 2. Velocity uses IMPACT's front-end, and the resulting IMPACT's Lcode is then translated into Velocity's X code IR. All traditional code optimizations are performed in Velocity, as well as some Itanium 2 specific optimizations. Our GMT list scheduling was performed after traditional optimizations, before the code is translated to Itanium 2's assembly, where Itanium 2-specific optimizations are performed, followed by register allocation and the final instruction scheduling pass. Velocity is parameterized with respect to the number of cores in the target processor.

To evaluate the performance of the code generated by Velocity, we used a validated cycle-accurate Itanium 2 processor [12] performance model (IPC accurate to within 6% of real hardware for benchmarks measured [20]) to build a CMP model comprising two Itanium 2 cores connected by the *synchronization array* communication mechanism proposed in [22]. Table 1 provides details about the simulator model. The simulator was built using the Liberty Simulation Environment [27].

The synchronization array (SA) in the model works as a set of low-latency queues. In our implementation, there is a total of 256 queues, each one with 32 elements. The SA has a 1-cycle access latency and has four request ports that are shared between the two cores. The IA-64 ISA was extended with `produce` and `consume` instructions for inter-thread communication. These instructions use the M pipeline, which is also used by memory instructions. This imposes the limit that only 4 of these instructions (minus any other memory instructions) can be issued per cycle on each core, since the Itanium 2 can issue only four M-type instructions in a given cycle. While the `consume` instructions can access the SA speculatively, the `produce` instructions write to the SA only on commit. As long as the SA queue is not empty, a `consume` and its dependent instructions can execute in back-to-back cycles.

The highly-detailed nature of the validated Itanium 2 model prevented whole program simulation. Instead, detailed simulations were restricted to the functions in question in each benchmark. We fast-forwarded through the remaining sections of the program while keeping the caches and branch predictors warm.

Benchmark	Function	Exec. %
adpcmdec	adpcm_decoder	100
adpcmenc	adpcm_coder	100
ks	FindMaxGpAndSwap	100
mpeg2enc	dist1	58
177.mesa	general_textured_triangle	32
179.art	match	49
300.twolf	new_dbox_a	30
435.gromacs	inl1130	75

Table 2. Selected benchmark functions.

To demonstrate the potential of our GMT scheduling technique, we applied it to important functions of selected applications from the MediaBench, SPEC-CPU, and Pointer-Intensive benchmark suites. Table 2 lists the selected application functions along with their corresponding benchmark execution percentages.

Figure 7 presents the speedup for the selected benchmark functions. For each benchmark, the two bars illustrate the speedup achieved with 2 and 10 cycles for the inter-core communication latency. With a 2-cycle communication latency, the speedups vary from 3.8% for adpcmenc to 173.3% for 435.gromacs, with a geometric mean of 38.7%.

The 2.7x speedup on two threads for 435.gromacs came as a surprise. The doubly nested loops in function `inl1130` contains an enormous amount of floating-point operations. We verified that, in the single-threaded version, this code suffers from a large number of spills of floating-point registers during register allocation (using graph coloring). In the multi-threaded version, the availability of twice as many registers enabled a much smaller number of spills, thus resulting in a reduced schedule height. Even though we just observed this advantage in one benchmark, we believe that this usage of additional resources will be even more beneficial in CMPs with smaller cores.

The results in Figure 7 show that, increasing the communication latency from 2 to 10 cycles, the geometric mean of the speedup drops from 38.7% to 31.8%. Additionally, we notice that the sensitiveness to the increased communication latency varies form benchmark to benchmark. In general, we noticed that functions with outer loops that iterate very fast (i.e. with small loop bodies), such the ones from adpcmdec and mpeg2enc, are more affected by the increased inter-core communication latencies. This is because, for such smaller loops, the communication latency corresponds to a larger percentage of the time necessary to execute one iteration of the loop.

We also conducted experiments to measure how sensitive the parallelized codes are to the size of the communication queues. Figure 8 shows the resulting speedups on our base model, with 32-element queues, and with the size

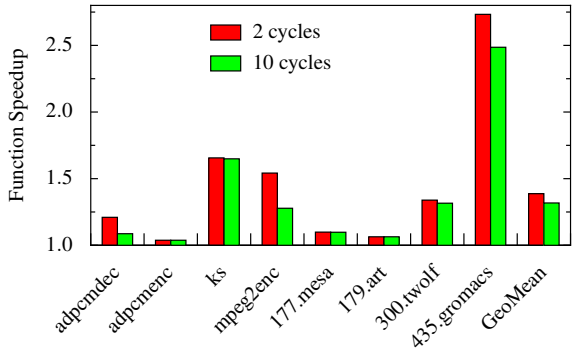


Figure 7. Speedup over single-threaded, for different inter-core communication latencies.

of the queues set to 1 element. The experiments show that only one of the benchmarks, ks, is affected by the reduced queue sizes. Investigation of the generated codes showed that, although the algorithms presented here may generate acyclic multi-threading such as DSWP [19], this was not the case in general. In fact, this was only observed for one inner loop in the ks benchmark. All other generated codes have cyclic multi-threading, in which pairs of cyclically dependent threads will always be less than one iteration apart. This explains why the benchmarks parallelized here are more susceptible to longer inter-thread latencies than the ones generated by DSWP [19]. The good side of this is that a cheaper inter-core communication mechanism, with simple blocking registers, is enough here.

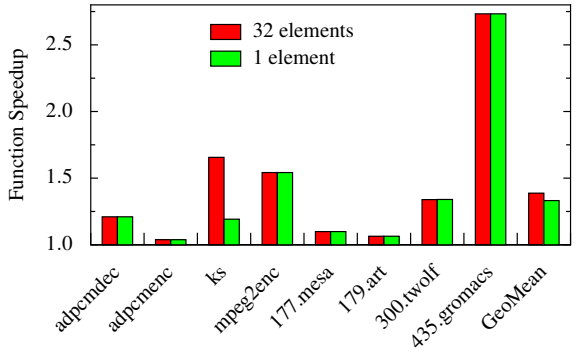


Figure 8. Speedup over single-threaded, for different communication queue sizes.

6 Related Work

There is a broad range of related work on instruction scheduling. In this section, we briefly describe and contrast the techniques mostly related to ours. The techniques are classified using a unified taxonomy that includes two orthogonal characteristics.

6.1 Local *versus* Global Scheduling

Instruction schedulers can be classified as either *local* or *global*. *Local* techniques independently schedule each straight-line sequence of instructions, typically a basic block. For each basic block, the instructions are scheduled respecting a *Data Dependence Graph* (DDG), where each vertex corresponds to an instruction and the arcs determine a partial ordering that must be respected in order to keep the correct program behavior. A classic example of local scheduling is *local list scheduling* [17], used in many optimizing compilers.

On the other hand, global schedulers simultaneously consider instructions from different basic blocks when making their schedule decisions. The set of basic blocks scheduled simultaneously can have different characteristics. For example, some techniques consider only instructions in basic blocks that form a trace in the CFG [7, 11]. Others schedule all the instructions in a set of basic blocks that form a loop in the program [14, 19]. The more general techniques must be able to simultaneously scheduled instructions in arbitrary CFG regions, potentially including the whole procedure. The special case of simultaneously scheduling instructions from control-equivalent basic blocks has been studied in [2]. A more general approach, based on integer linear programming and combining scheduling and global code motion, was proposed in [28]. Compared to local approaches, global schedulers use a larger scope to help them making decisions, and thus have potential to obtain a better schedule. Besides data dependences, global schedulers must also preserve control dependences.

6.2 Single- *versus* Multi-Threaded Scheduling

Depending on the number of simultaneously executing threads they generate, scheduling techniques can be classified as either *single-threaded* or *multi-threaded*. Of course, this characteristic is highly dependent on the target architecture. Single-threaded scheduling is commonly used for a wide range of single-threaded architectures, from simple RISC-like processors to very complex ones such as VLIW/EPIC [14, 3] and clustered architectures [5, 18].

Besides scheduling original program's instructions (the *computation*), multi-threaded schedulers must also generate *communication* instructions to satisfy inter-thread dependences. It is true that, for clustered single-threaded architectures, the scheduler also needs to insert communication instructions to move values from one register bank to another. However, the fact that dependent instructions are executed in different threads makes the generation of communication more challenging for multi-threaded architectures.

Motivated by CMPs, several multi-threaded scheduling techniques have recently been proposed to generate multi-threaded code from general-purpose, sequential applica-

Num. of Threads	Scope			
	Basic Block	Trace	Loop	Proc.
Single	List Sched. [17]	Trace [7, 5, 3] Superblock [11]	SWP [14, 18]	GSTIS [2] ILP [28]
Multiple	Space-time [15] Convergent [16] DAE Sched. [23]		DSWP [19]	*GMT*

Table 3. Instruction scheduling space.

tions [15, 16, 19]. Most of these techniques use a *local multi-threaded* (LMT) approach [15, 16]. LMT schedulers have to insert synchronization at branch instructions: before jumping to the next block, the thread taking the branch decision sends the branch direction to the other threads through the communication queues. The other threads then mimic this branch, so that all threads follow the same path through the program's control-flow graph (CFG) [15, 16]. A similar approach is used by schedulers for decoupled access/execute architectures, which may even use specialized queues to communicate branch directions [23].

Although suitable for single-threaded architectures, the problem of using local scheduling for multi-threaded machines is that it effectively only exploits *instruction-level parallelism* inside basic blocks. Unfortunately, general-purpose applications typically have a very small number of instructions per basic block, usually less than 10. Not surprisingly, existing compilers based on local scheduling for multi-threaded architectures have shown little effectiveness in extracting parallelism from general-purpose, sequential applications [15, 16].

In [19], we recently proposed a global multi-threaded scheduling technique, called decoupled software pipelining (DSWP). This technique utilizes separate threads to execute different stages of a loop in a pipelined fashion. Although DSWP is classified as global scheduling, it is limited to loop regions. Besides that, the technique described in [19] imposes some extraneous loop dependences, which we later proved unnecessary.

In this paper, we presented general *global multi-threaded* scheduling algorithms, which can simultaneous schedule instructions in arbitrary code regions. Table 3 summarizes how various existing scheduling techniques are classified according to our taxonomy. Horizontally, the more a techniques is to the right, the more general is its handling of control flow.

7 Conclusion

The recent trend in the microprocessor industry to build chip multiprocessors (CMPs) has increased the interest in automatic thread extraction for the large base of non-scientific applications. Despite this interest and CMPs' ability to simultaneously execute different control paths, existing multi-threading techniques have mostly been restricted to local scheduling approaches. This paper introduced the concept of global multi-threaded (GMT) instruction scheduling, a general technique that can exploit fine-grained thread-level parallelism on modern CMPs. Opposed to local approaches, GMT scheduling exposes thread-level parallelism, enabling the concurrent execution of instructions from different regions of the control-flow graph. This paper also described algorithms to find profitable GMT schedules for arbitrary code regions using a PDG representation, as well as an algorithm to generate multi-threaded code for any GMT schedule decision. Experimental results on a number of benchmarks demonstrated the enormous potential of our techniques.

References

[1] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated predication and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 227–237, June 1998.

[2] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 241–255, June 1991.

[3] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 180–192, April 1987.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[5] J. Ellis. *Bulldog: A Compiler for VLIW Architectures*. The MIT Press, Cambridge, MA, 1985.

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, July 1987.

[7] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W H Freeman & Co, New York, NY, 1979.

[9] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[10] P. Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Language Systems*, 19(4):557–567, 1997.

[11] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.

[12] Intel Corporation. *Intel Itanium 2 Processor Reference Manual: For Software Development and Optimization*. Santa Clara, CA, 2002.

[13] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[14] M. S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 318–328, June 1988.

[15] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. P. Amarasinghe. Space-time scheduling of instruction-level parallelism on a Raw Machine. In *The Proceedings of the Eighth International Confrence on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, 1998.

[16] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe. Convergent scheduling. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, November 2002.

[17] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan-Kaufmann Publishers, San Francisco, CA, 1997.

[18] E. Nystrom and A. E. Eichenberger. Effective cluster assignment for modulo scheduling. In *Proceedings of the 31st International Symposium on Microarchitecture*, pages 103–114, December 1998.

[19] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture*, November 2005.

[20] D. A. Penry, M. Vachharajani, and D. I. August. Rapid development of a flexible validated processor model. In *Proceedings of the 2005 Workshop on Modeling, Benchmarking, and Simulation*, June 2005.

[21] R. Rangan, N. Vachharajani, A. Stoler, G. Ottoni, D. I. August, and G. Z. N. Cai. Support for high-frequency streaming in cmps. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 259–269, December 2006.

[22] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 177–188, September 2004.

[23] K. Rich and M. Farrens. Code partitioning in decoupled compilers. In *Proceedings of the 6th European Conference on Parallel Processing*, pages 1008–1017, Munich, Germany, September 2000.

[24] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[25] V. Sarkar. A concurrent execution semantics for parallel program graphs and program dependence graphs. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, 1992.

[26] M. B. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal. Scalar operand networks. *IEEE Transactions on Parallel and Distributed Systems*, 16(2):145–162, February 2005.

[27] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 271–282, November 2002.

[28] S. Winkel. Exploring the performance potential of Itanium processors with ILP-based scheduling. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2004.

[29] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, September 1994.

Optimal placement of fused multiply-add (FMA) instructions.

Konstantin Serebryany
konstantin.s.serebryany@intel.com
14, Bolshoy Savvinisky per., Moscow 119435, Russia

Abstract

Many modern computer architectures have so called 'fused multiply-add' instructions (FMA), which compute `a*b+c` as a single operation. An optimizing compiler for such computer architecture should perform an optimization which combines additions and multiplications into FMAs. This article introduces a method for combining floating-point operations into the optimal sequences of FMA instructions. The method is based on pattern generation and pattern matching.

Introduction

Several modern computer architectures, including Itanium® and PowerPC®, feature so called *fused multiply-add (FMA)* instructions which compute ternary floating-point expressions as a single operation ([1]):

`FMA(a,b,c)  =  a*b+c`; (FMADD in PowerPC®)

`FMS(a,b,c)  =  a*b-c`; (FMSUB in PowerPC®)

`FNMA(a,b,c) = -a*b+c`; (FNMSUB in PowerPC®)

`FNMS(a,b,c) = -a*b-c`; (FNMADD in PowerPC®, not present in Itanium®)

For such architectures it is critical to use FMA instructions, especially for computations that contain a mix of floating point additions, subtractions and multiplications. Numerous successful attempts have been made to optimize some specific codes for FMA architectures (see 'Related work' below).

Most optimizing compilers, however, use simple 'peephole' techniques to fuse multiplies and adds. For example, the GNU compiler (gcc) targeted to Itanium® generates suboptimal FMA sequences for some floating point expressions. For C code: `res=-a*(b*c-d)-e;` gcc 4.0.0 generates `t1=FMS(b,c,d); t2=FMS(0,0,t1); res=FMS(a,t2,e)`, while the best possible sequence is `t1=FNMA(b,c,d); res=FMS(a,t1,d)`.

In this paper we present an algorithm which finds optimal FMA sequences for floating point expressions containing multiplications, additions, subtractions and negations.

In section 1 we define what is an 'optimal' FMA sequence. In section 2 we introduce the concept of an FMA pattern as well as an algorithm to generate a table of all possible FMA patterns of a given size. In section 3 we outline a pattern matching algorithm, which matches an incoming expression against the table of FMA patterns. Section 4 summarizes practical results of applying this algorithm.

Related work

We are not aware of any prior work that allows finding optimal FMA sequences for arbitrary floating point expression during compile time. However, this paper uses very well known approach which is used in a number of areas:

- In [2] Harrison *et al* generate latency-optimal FMA sequences for polynomials of special kind.
- In [3] and preceding papers the authors describe methods of minimizing the number of FMAs in DFT computations.
- In [4] Briggs and Harvey use search to minimize the cost of integer multiplication by constant.

1. The definition of optimality

By '*optimal* instruction sequence', one usually means a sequence which has the minimal possible value of some particular cost function.

The simplest cost function which could be applied to FMA sequences is the number of instructions in the sequence (**complexity criteria**). With this cost function the sequence `t1=FNMA(b,c,d); res=FMS(a,t1,d)` will be optimal for the expression `res=-a*(b*c-d)-e;` (since there exists no equivalent sequence of smaller size).

Another frequently used cost function is the height of the DAG of the sequence (**latency criteria**). For example, if we want to optimize a function

```
double abcd(double a, double b, double c, double d){
    return a * b * c * d;
}
```

the optimal sequence will be `t1=FMA(a,b,0); t2=FMA(c,d,0); res=FMA(t1,t2,0)` since it has the minimal possible height (and latency).

In some cases these two cost functions may conflict with each other. For example, the latency-optimal sequence for the expression `a*b+c*d+e*f+g*h` will have height 3 and complexity 5, while the minimal

complexity 4 is achievable only with height 4. Figure 1 shows these two sequences as acyclic directed graphs (DAGs).

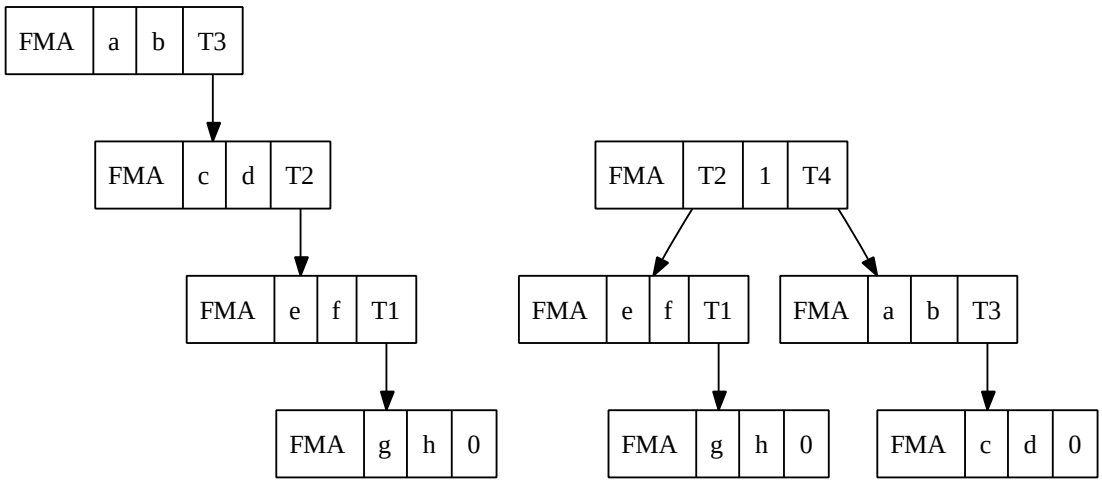


Figure 1: Two ways to generate an FMA sequence for `a*b+c*d+e*f+g*h`.

In some cases compiler has to analyze more than just an expression itself, but also the nature of the operands of the expression.

```
double abc(double a, double b, double *p){
    double c = *p;   return a * (b * c);
}
```

In the above function the first two arguments of the expression `a * (b * c)` are already in a register, while the third argument has to be loaded from memory. So, the sequence `T1=FMA(a,b,0); RES=FMA(T1,c,0)` is better than `T1=FMA(b,c,0); RES=FMA(T1,a,0)`, since in the first sequence the result of the memory load `*p` is used later. We call this condition the **argument availability criteria**.

It is obvious that a good cost function must combine at least these three criteria, but which of these three criteria is more important is not so obvious. However, some basic heuristics may be used:

- If the expression is in a loop, which is going to be software-pipelined (SWP-loop), the latency is of less importance.
- In SWP-loop, argument availability is important only for those arguments that are defined by a previous iteration.

- If a basic block is not in an SWP-loop and has only one expression, the complexity is of less importance.

A more detailed analysis of the cost function is outside of the scope of this article. To simplify our explanation we will assume, for the rest of the article, the following definition of optimality.

An FMA DAG represents an optimal FMA sequence for a given expression if

1. **[Minimal complexity]**: the number of instructions in the DAG is minimal.
2. **[Minimal latency]**: the height of this DAG should be minimal compared to all possible DAGs with minimal complexity.
3. **[Argument availability]**: if a strict order is defined on the set of terminals (arguments), smaller terminals should be placed as close to the root node as possible, while preserving minimal complexity and latency. If some terminals are available later than other terminals, this rule allows us to use late terminals later (closer to root node of DAG).

2. Pattern generation

We shall say that a sequence of N FMA (not FMS nor FNMA) instructions $F_1, F_2, ...F_N$ is an **FMA pattern** if

- Instruction F_i may have these operands: instruction $F_j, j > i$, terminal $A, B, ...$, constant 0 or 1.
- The sequence forms a connected DAG with one root node (F_1).
- Each of the terminals $A, B, ...$ appear only once in the sequence and smaller (lexicographically) terminals appear first; if a sequence has M terminals then these are the first M letters of alphabet.

We define the **canonical form** of an FMA pattern as an expression resulting from 'opening all parentheses' in the pattern and ordering all the summands lexicographically.

The **shape** of an FMA pattern is defined as an integer number, computed from the canonical form using the following rules: a) replace all terminals in the canonical form with '1' and all '+' characters with '0', b) interpret the resulting string as an integer in the binary form.

Examples:

FMA pattern	canonical form	shape
`F1=FMA(A,B,C)`	`AB+C`	`1101`
`F1=FMA(A,B,F2);` `F2=FMA(C,D,0)`	`AB+CD`	`11011`

`F1=FMA(F2,F2,F3);`
`F2=FMA(A,1,B);`
`F3=FMA(C,D,E)`
	`AA+AB+AB+BB+CD+E`	`1101101101101101`

The number of possible FMA patterns of size N raises exponentially. Thus, even if we forbid constants 0 and 1 as arguments, the number of FMA patterns for $N = 1$ is 1, for $N = 2$ is 5, for $N = 3$ is 67, for $N = 4$ is 1877 and for $N = 5$ is 94891. With constants 0 and 1 these numbers are even larger. Of course, not all patterns may represent an optimal FMA sequence. For example, the pattern `F1=FMA(A,1,0)` is redundant because it represents a single terminal `A`. Some patterns may not be considered for practical reasons, e.g. the pattern `F1=FMA(F2,F2,F2); F2=FMA(F3,F3,F3); F3=FMA(A,B,C)` will hardly be matched in a real program.

Since the table of FMA patterns should be generated only once, the speed of pattern generation is not critical. The following is an outline of the algorithm we used to generate a table of FMA patterns of size 5 and less. It takes few minutes on a modern CPU to generate a table of approximately 100000 entries.

- Generate all possible FMA patterns of size 5 or less (simple recursive algorithm) but prune those patterns that are either redundant or unlikely to appear in real life (only for size 5; for size 4 and less we may generate all patterns; the process of removing redundant patterns is automatic and uses simple heuristics, e.g. not less than 3 terminals should be present in the 5-fma pattern).
- Compute the canonical forms and the shapes of the patterns.
- Sort the patterns according to shape.
- Within each shape sort patterns according to size (the number of FMAs, the smaller go first).
- Within each group of patterns with the same size and shape, sort the patterns according to the DAG height (patterns with smaller height go first).

Each table entry may be represented as a string or may be compressed into a 64-bit integer. The generated table may be stored as a separate text file or linked to a compiler as an array of data.

3. Pattern matching

The task of pattern matching may be formulated as follows:

Given: A pre-generated table of FMA patterns and an expression consisting of operations `-` (unary and binary), `*`, `+`, parentheses and terminals `a`, `b`, `c`, ... (the terminals are ordered according to their availability)

Find: An equivalent optimal sequence of FMA, FMS, FNMA (and FNMS, if applicable) instructions.

First of all, the canonical form and the shape of the expression should be found (again, we 'open parentheses' and sort summands lexicographically). This time, however, the canonical form may contain summands with sign `-` (it is replaced with 0 to compute the shape).

Second, we choose those patterns from the table which have the same shape as the incoming expression and contain at least as many terminals as the incoming expression.

Then, for each chosen pattern we try to find a **valid mapping** between formal and actual terminals and a **valid sign combination**. A mapping between formals (`A`, `B`, `C`, ...) and actuals (`a`, `b`, `c`, ...) is valid, if replacing formals with actuals in the FMA pattern and, next, computing the canonical form lead to the same canonical form as the canonical form of the incoming expression with '`-`' signs replaced with '`+`'. The valid sign combination is such a sequence of FMA, FMS, FNMA (and FNMS, if applicable) instructions, that, if we replace the FMA instructions in the pattern with this sequence, apply the valid mapping of terminals and compute the canonical form we will get the same canonical form as we have got for the incoming expression. Figure 2 shows an example of such pattern matching. In this example the pattern matching algorithm achieves the effects of *common subexpression elimination (CSE)* by eliminating redundant evaluations of `b*b`, *factorization* by factorizing another instance of `b` and *tree height reduction* by creating a DAG with the minimal possible height.

The algorithm stops when the first valid mapping and the first valid sign combination are found. Since the patterns are sorted according to the minimal complexity and the minimal height, the matched DAG is optimal.

The only computationally-intensive part of the algorithm is the process of finding the valid mapping (see section 3.1). Valid sign combinations can be found by a simple exhaustive search (for 5 instructions there are only 3^5 possible sign combinations if FNMS instruction is not present and 4^5 otherwise).

3.1 Finding a valid mapping

Suppose we have an incoming expression with N_a actual terminals $A_1, A_2, ..., A_{N_a}$ and an FMA pattern with N_f formal terminals $F_1, F2, ..., F_{N_f}$. The shape of the incoming expression and the shape of the pattern are the same and $N_f \geq N_a$. It is possible to find all the mappings between the formals and the actuals and, for each

`e*(-abbbbb+bbbc-bd)`	incoming expression.
`-abbbbbe+bbbce-bde`	Canonical form of incoming expression (`a`, `b`, ... `e` are the actual terminals).

`F1=FMA(F2,F4,0); F2=FMA(F3,F5,A);`
`F3=FMA(F5,B,C); F4=FMA(D,E,0);`
`F5=FMA(F,G,0)`
	One of the FMA patterns in the table (`A`, `B`, ... `G` are the formal terminals).

`+BDEFFGG+CDEFG+ADE`	Pre-computed canonical form of the pattern (it has the same shape as the incoming expression).
`A->d;B->a;C->c;D->b;E->e;F->b;G->b;`	Valid mapping (formal->actual).

`F1=FNMA(F2,F4,0); F2=FMA(F3,F5,d);`
`F3=FMS(F5,a,c); F4=FMA(b,e,0);`
`F5=FMA(b,b,0);`
	The resulting DAG with actual terminals and valid sign combination.

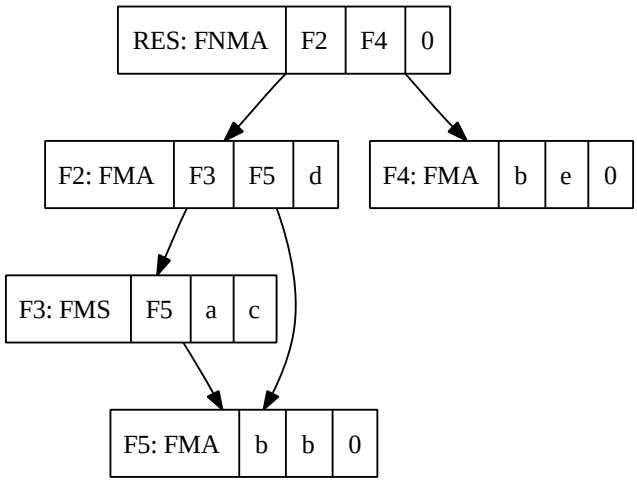


Figure 2: An example of pattern matching

mapping, to check whether or not it is valid. However, the number of all the mappings is ${N_f}^{N_a}$, which may be too large.

We used a recursive breadth-first search combined with several constraints, which allows to cut the search tree early. This algorithm is quite similar to the breadth-first search used to solve the '8 queens' problem ([5]). The outline of the algorithm in a pseudo code is given in the Appendix.

The algorithm assumes that the actual terminals are sorted according to their availability, i.e. the terminal A_i will be available not sooner than A_j if $i > j$. Similarly, the FMA patterns are generated in such a way that the formal terminal F_i is not farther from the rot node than F_j if $i > j$. As the first actual terminals are tried first during the pattern matching and as the recursive search starts from the first formal terminal, the **argument availability** criteria is satisfied (see section 1).

This algorithm may be *exponential in the worst case*, i.e. in the case when no early cuts of the search tree happen and when the matching fails after trying all possible permutations of formal terminals. A 5-fma pattern may contain up to 11 formal terminals, and the number of all possible permutations of formal terminals may be huge. In practice we found only one such case: `-a-b-c-d-e-f`. Trying to match this expression against the 5-fma pattern `A+B+C+D+E+F` the matcher will search all the possible permutations of formal parameters (which is $6! = 720$) and will not find a valid mapping because on Itanium® `-a-b-c-d-e-f` requires 6 fma operations (for an architecture with FNMS the matching algorithm will find the valid mapping immediately).

Since we have proved (by an example) the exponential complexity of the matching algorithm, we have implemented a limit in our matcher – the matcher will stop after 1000 attempts to find a valid sign combination. However, in practice this limit was never reached during our testing[1]. More than 99% of matches finished after less than 50 attempts.

3.2 The proof of optimality

The optimality (according to the definition in section 1) of the generated FMA sequence is guaranteed only[2] if the sequence consists of 5 or less FMAs (i.e. the expression has been matched against some pattern in the table). For large expressions the optimizer will recursively optimize subexpressions.

The minimal complexity and the minimal latency of the generated sequence are guaranteed by the fact that the matching algorithm starts the search from the patterns with the least number of FMAs and the least height (see section 2).

The criteria of argument availability is satisfied because of these facts: first, the formal parameters are ordered in such a way that the smaller parameters are closer to the DAG root; second, the actual parameters are ordered according to their availability (the terminals available later go first). The matching algorithm (see Appendix) begins with trying to match the first formal parameter to the first actual parameter. If there exists a valid mapping such that the first formal parameter `A` is mapped to the first actual parameter `a`, then the terminal `a` (which is available later than others) will be used closer to the root node (i.e. later) than others. If there is a valid mapping at all, the terminal `a` will be mapped to the formal parameter with the smallest possible number. The same logic is valid for all other actual parameters.

[1] While experimenting with 6-fma and 7-fma patterns, this limit was reached on less than 0.01% of expressions

[2] Of course, if the optimal pattern for a given expression was pruned during the pattern generation, the generated sequence will not be optimal either.

4. The results

The presented algorithm is implemented in the Intel® C++ and Intel® Fortran compilers for Itanium®, starting from the version 9.0. Since the floating point transformations performed by this algorithm may significantly change the accuracy of computed expressions[3], the new transformation is enabled only under a compiler option `-IPF-fp-relaxed`. The performance improvements we observed on benchmarks from the CPU2000 and CPU2006 suites (http://www.spec.org) are summarized in figure 3. The performance was measured with flags `-O3 -ipo -IPF-fp-relaxed`[4] with and without the new phase. In the hot spots which have improved the most significantly we observed factorization, CSE and as well as tree height reduction *simultaneously*.

Benchmark	Performance gain
173.applu	11%
435.gromacs	9%
436.cactusADM	3%
454.calculix	3%

Figure 3: Performance improvements on CPU2000 and CPU2006.

Even though the FMA pattern matching algorithm is computationally intensive, we have not observed overall compile-time increase of more than 1% (as measured on CPU2000 and CPU2006 benchmarks and other tests).

Our experiments with 6-fma and 7-fma patterns did not show additional performance increase on SPEC benchmarks.

Conclusions

In this paper we have presented a new algorithm for the optimal placement of fused multiply-add instructions (FMA). The algorithm can handle only small and medium-sized expressions, because it uses pattern matching and a pre-generated table of patterns of a limited size. The current implementation handles expressions that may be possibly represented as 5 or less FMAs, but larger expressions could be handled in expense of larger table size and increased compile-time. An expression of an arbitrary size can be optimized as well, if the algorithm is applied to its sub-expressions repeatedly. The presented algorithm not only fuses multiplies and adds optimally, but also achieves the effects of *common subexpression elimination (CSE)*, *factorization* and *tree-height reduction*.

[3] The described aggressive transformations of floating point expressions do affect the accuracy of computations, but in our practice the most common source of problems with precision is the transformation `a*b+c*d ==> FMA(a,b,c*d)`, which is performed by most compilers without special options. Nevertheless, with the new algorithm we have not observed accuracy problems except for some tests that are already well known for floating point accuracy issues.

[4] A simple FMA peephole is enabled in the Intel® compiler at all optimization levels starting from -O1

This new algorithm has a practical importance for computer architectures featuring FMA instructions — it allows getting up to 11% speedup (as measured on CPU2000 and CPU2006 benchmarks).

The algorithm is limited to the FMA instructions, but the same approach may be used to handle other groups of instructions on any computer architecture, e.g. to perform optimal factorization+CSE+THR for integer expression with operations `+,-,*`. For example, the following equivalence may be easily found by pattern matching: `c*d+a*b*d+a*b*c+a*a*b*b ==> (c+ab)*(d+ab) ==> t=a*b;(c+t)*(d+t)`.

Appendix: The recursive breadth-first search with constraints.

```
// Try to map i-th formal. Should be called as TRY(0)
// NF -- number of formals, NA -- number of actuals.
void TRY(int i)
{
    if(i == 0) {/* clear the mapping*/}
    // at this point we mapped first i formals: 0, 1, ... i-1
    if(i == NF) {
        // We mapped NF formals, i.e a full mapping is found.
        // Replace terminals in the DAG using this mapping.
        // Try all 3^complexity sign combinations in the DAG.
        // If with some sign combination the canonical form
        // of the dag is equal
        // to the incoming canonical form, we found a valid mapping
        // and sign combination: stop searching.
        return;
    }
    // At this point we have to decide whether we want to continue
    // with this partial mapping.
    if(!PARTIAL_MAPPING_IS_GOOD()){
        return;
    }
    // try to map i-th formal to each actual [0..NA)
    // The order is essential: it guaranties that terminals
    // available later will be used later.
    for(int a = 0; a < NA; a++){
```

```
        // update the mapping: map i-th formal to 'a'
        TRY(i+1);
    }
}
// We have a partial mapping between formals and actuals.
// Return false if we can prove that this partial mapping
// can not be a part of valid mapping for the given formal
// and actual canonical forms.
bool PARTIAL_MAPPING_IS_GOOD()
{//A number of constraints can be computed for each terminal,
// e.g. maximal/minimal power of terminal in expression,
// number of products in which the terminal is used, set of valid neighbors
// (terminals used in the products where this terminal is used), etc.
// If the partial mapping contradicts any of these constraints,
// return false.
}
```

References

[1] http://en.wikipedia.org/wiki/Fused_multiply-add

[2] John Harrison, Ted Kubaska, Shane Story and Ping Tak Peter Tang, "The Computation of Transcendental Functions on the IA-64Architecture", http://developer.intel.com/technology/itj/q41999/pdf/transendental.pdf

[3] Yevgen Voronenko and Markus Pueschel, "Automatic Generation Of Implementations For DSP Transforms On Fused Multiply-Add Architectures", http://citeseer.ist.psu.edu/voronenko04automatic.html

[4] Preston Briggs and Tim Harvey, Multiplication by integer constants, citeseer.ist.psu.edu/briggs94multiplication.html

[5] http://en.wikipedia.org/wiki/Eight_queens_puzzle

[6] Herbert Karner, Martin Auer and Christoph W. Ueberhuber, "Accelerating FFTW by Multiply-Add Optimization", http://citeseer.ist.psu.edu/273715.html

[7] Stef Graillat, Philippe Langlois, Nicolas Louvet, "Improving the compensated Horner scheme with a fused multiply and add", SAC '06: Proceedings of the 2006 ACM symposium on Applied computing, 2006.

A Practical and Complete Implementation of SSUPRE without Static Single Use Representation

Yao Shi
Tsinghua University
shiyao00@mails.tsinghua.edu.cn

Tianwei Sheng
Tsinghua University
ctw04@mails.tsinghua.edu.cn

Hucheng Zhou
Tsinghua University
hc_zhou@nudt.edu.cn

Dehao Chen
Tsinghua University
chendh05@mails.tsinghua.edu.cn

Shinming Liu
Hewlett-Packard
shin@cup.hp.com

Wenguang Chen
Tsinghua University
cwg@tsinghua.edu.cn

Weimin Zheng
Tsinghua University
zwm-dcs@tsinghua.edu.cn

Abstract

SSUPRE is a sparse algorithm for partial store redundancy elimination that is the dual problem to partial expression redundancy elimination that is solved by the well known SSAPRE algorithm. SSUPRE uses static single use (SSU) representation that is not used for other optimizations. The complexity of construction of SSU representation is equal to that of SSA, which is $O(v^2)$. The new SSUPRE algorithm we implemented in Open64 Compiler can efficiently handle both scalar stores and indirect stores without expensive construction of SSU representation. It is linear in complexity.

Keywords Partial Redundancy Elimination, Static Single Use, Factored Redundancy Graph

1. Introduction

Partial redundancy elimination (PRE) is an important technique for code optimization. Partial redundancy elimination includes elimination of partial redundancy, common sub-expression (CSE) and loop invariant hoist [1] shown as Figure 1. The traditional PRE solutions based on bit vectors are dense so that they are being replaced by sparse implementation in most production compilers.

SSAPRE [3] is a sparse solution put forward in 1997. It can handle partial expression redundancy elimination (EPRE) but cannot handle partial store redundancy elimination (SPRE) shown as the right part of Figure 2. As a dual problem to EPRE, SPRE is solved by SSUPRE algorithm [5] which is dual to SSAPRE algorithm.

SSUPRE is first implemented in Silicon Graphics MIPSpro Compiler 7.2[5], which is the preexistence of Open64 Compiler [2]. The static single use (SSU) representation is the input of SSUPRE algorithm of this implementation [5]. But SSU representation is expensive to construct (the complexity is same as SSA construction, i.e. $O(v^2)$, where v is the number of SSU graph nodes) and is not practical because that it cannot benefit other optimizations. In theory, the complexity can be lower to $O(e)$ at the cost of more complicated algorithm like SSA construction [11] [12], where e is the number of SSU graph edges.

In the original SSUPRE implementation, only scalar stores is processed due the heavy overhead of constructing a complete SSU representation. The original developers only implemented a simplified SSU representation

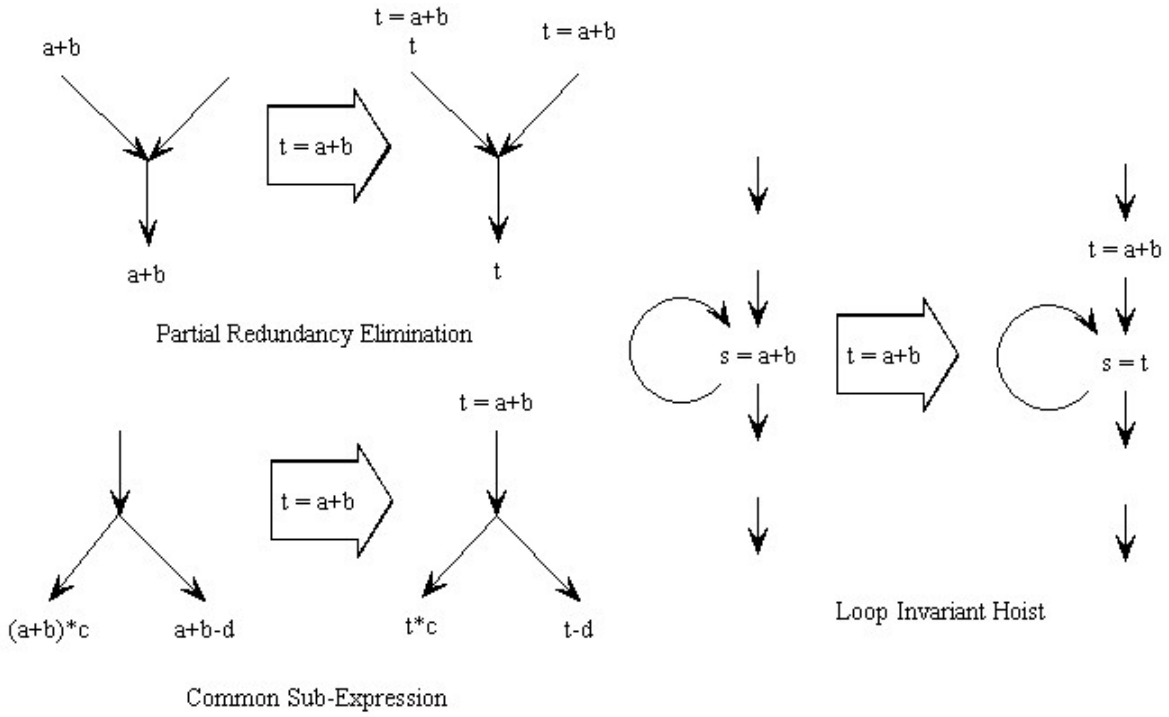


Figure 1. Partial redundancy elimination.

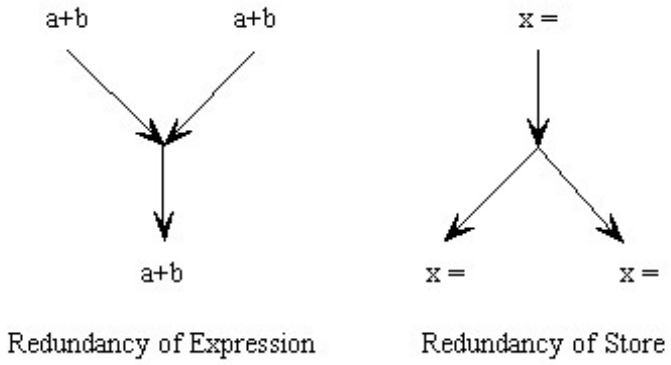


Figure 2. Duality of redundancy.

for their SSUPRE. This SSU representation cannot handle the store with left hand side of recursive expression, i.e. $*p \leftarrow < expr >$. Some other works like [4] do not process indirect stores and alias issues either.

However, it is important to process indirect stores in C++ programs especially because indirect loads and indirect stores are performed to access data members of classes frequently. For a well encapsulated C++ program, most of store statements in it's class methods are indirect stores. We have summarized the number of indirect stores in some cases of SPECCPU programs in Table 4.

We have implemented SSUPRE in Open64 Compiler without SSU representation , which can handle both scalar stores and indirect stores like $*p \leftarrow < expr >$. In other words, the store with left hand side of recursive expression can be processed by our new implementation cooperating with partial expression redundancy elimination. See Figure 3.

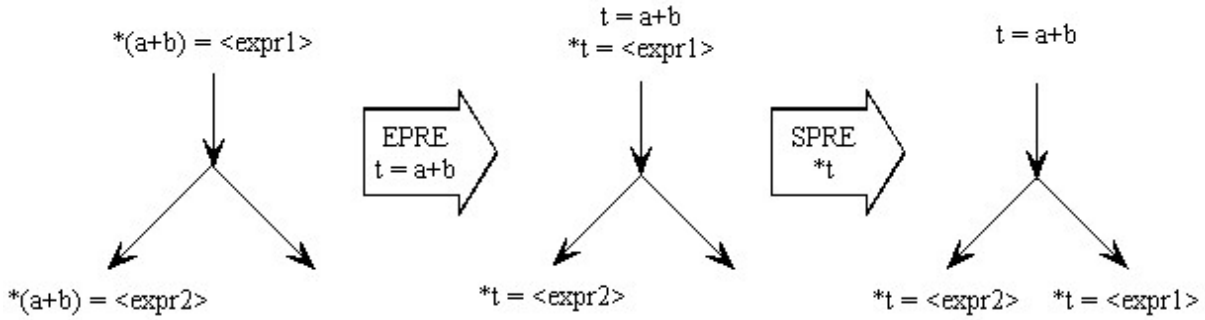


Figure 3. Partial redundancy elimination for left hand side of recursive expression.

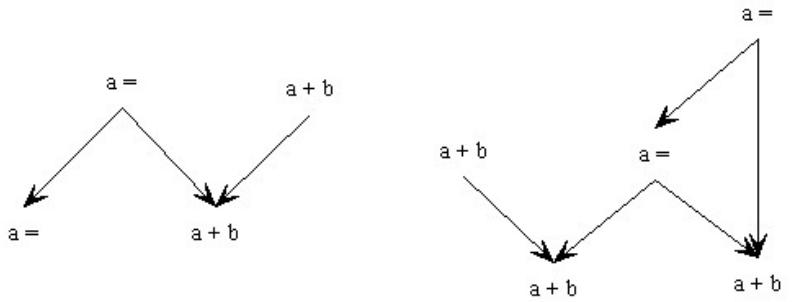


Figure 4. Cases that Morel's algorithm cannot process.

2. Related Works

2.1 Partial Store Redundancy Elimination

Partial redundancy elimination invented by Morel and Renvoise [1], shown as Figure 1, is an optimiation technique that unifies redundancy elimination, common sub-expression and loop invariant hoist. Morel's technique based on bidirectional data flow equations is complicated and is not computational optimal. That means it cannot handle the situation as Figure 4 shows.

Knoop et al. improved the original PRE algorithm and proofed that their algorithm is computational optimal [7]. Knoop's method named "lazy code motion" uses uni-directional equations instead of Morel's mysterious bidirectional equations and reduces register pressure by inserting computations as late as possible.

Lazy code motion is mature enough in theory. It can process the cases as Figure 4 that Morel's algorithm cannot do. But The time complexities of the algorithm are too high to compile large programs for production compilers because lazy code motion is still a dense solution based on bit-vector formulation.

Chow et al. presented a sparse algorithm named SSAPRE [3] based on static single assignment (SSA) [6] form that is a sparse representation for many efficient global optimizations. SSAPRE uses SSA-like method to handle the elimination of expression redundancy. Since SSAPRE cannot deal with partial store redundancy elimination (SPRE), SSUPRE algorithm that can solves SPRE was designed and implemented by the same group later on. Our work is based on the original SSUPRE algorithm.

2.2 Safety and Redundancy

In this subsection, we introduce some basic concepts of partial redundancy elimination.

For a computation at point S along a path P from the entry to S, it is *available* if there exists at least one computation along P. For a computation at point T along a path Q from T to the exit, it is *anticipated* if there exists at least one computation along Q.

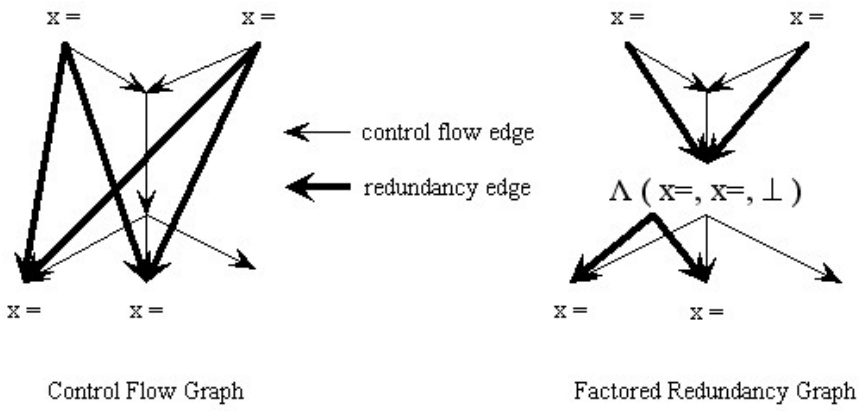


Figure 5. Factored redundancy graph for store PRE.

For a computation at point S, if there always exists at least one computation along each way from the entry to S, this computation is *fully available*. Otherwise, if the computation is available along some of these ways, it is called partial available.

Similarly, if there always exists at least one computation along each way from point T to the exit, the computation at point T is *fully anticipated*. Otherwise, if the computation is anticipated along some of these ways, it is called partial anticipated.

Fully available is also called *up-safe* and fully anticipated is called *down-safe*. Then we define that a placement is safe if the optimization at this place has not introduced any other new values to any path of the program. A placement is safe if it is up-safe or down-safe. [9].

In the view of redundancy, expressions are different from stores. A store is fully/partial redundant if it is fully/partial anticipated while an expression is fully/partial redundant if it is fully/partial available. The bottom "$a + b$" and the top "$x =$" are fully redundant in Figure 2. Obviously, fully redundancies can be eliminated directly. The basic idea of PRE algorithm is to insert some computations that make the original partial redundancies fully redundant. Then delete these fully redundancies.

2.3 Factored Redundancy Graph

The fundamental problem of sparse partial redundancy elimination is the sparse representation of expressions. A representation called factored redundancy graph (FRG) is described in SSAPRE [8]. We describe FRG in the view of partial store redundancy elimination though the original FRG is in the view of partial expression redundancy elimination.

The left part of Figure 5 is the dense representation of redundancies. In this graph, each real store may be the redundancy of more than one other store. This phenomenon is complicated and is hard to process. It must estimate whether all the occurrences which the real store is redundancy of are fully anticipated to determine whether the real store can be removed. The solution to the control flow graph is dense and its high complexity causes too long time for compiling.

In the right part of Figure 5, some Λs regarded as stores are inserted into the graph. These Λs cause that each real store is the redundancy of at most one corresponding occurrence. That means a real store can be deleted,i.e. the code motion is safe, if the unique occurrence which the real store is redundancy of can be a fully anticipated (down-safe) Λ after Λ-Insertion. This condition is much simpler than that of above dense graph.

2.4 Original SSUPRE

Original SSUPRE algorithm is made up of 6 phases like SSAPRE algorithm: Λ-Insertion, Renaming, UpSafety, WillBeAnt, Finalize and Code Motion. [5]

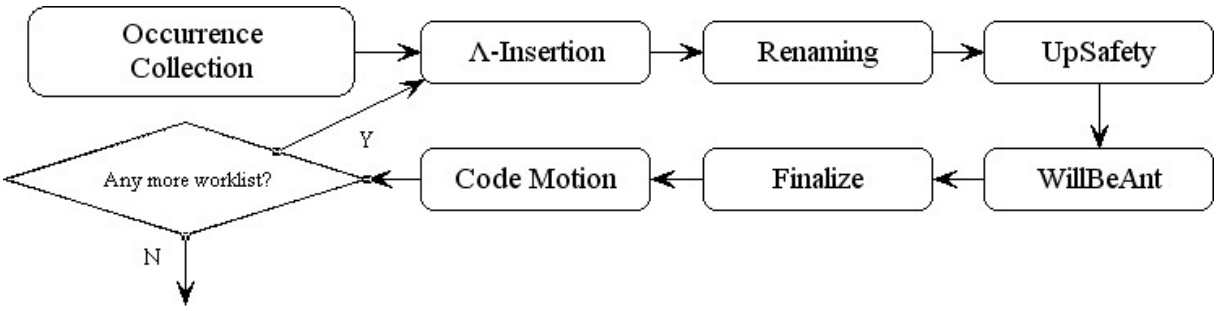


Figure 6. Process of partial store redundancy elimination.

Λ-Insertion phase inserts Λs in the control flow graph so that each real store occurrence is the redundancy of at most one corresponding occurrence (real store occurrence or Λ occurrence). Actually, it converts the control flow graph to factored redundancy graph mentioned in subsection 2.3.

Renaming phase sets the versions of every real store occurrence and Λ occurrence. Suppose that occurrence X and occurrence Y have the same version. If X post-dominant Y, Y is X's redundancy. Otherwise, X and Y must be the redundancies of the same occurrence if X and Y have not post-dominance relationship.

UpSafety phase checks whether the Λs are up-safe. Only redundancies of the Λs satisfy can_be_ant that is computed by UpSafety can be deleted because that can_be_ant is the necessary condition of code motion.

WillBeAnt phase sets the property will_be_ant that determines whether some stores are inserted to make a Λ fully anticipated so that the redundancies of this Λ can be deleted.

Finalize phase summarizes the information of insertion and deletion. For a Λ satisfies will_be_ant, extra stores should be inserted to make it fully anticipated. And then the redundancies of this Λ can be deleted. The last phase Code Motion performs the actual insertion and deletion.

3. SSUPRE Algorithm without SSU

As the subsection 2.4 mentioned, the original SSUPRE has six phases: Λ-Insertion, Renaming, UpSafety, WillBeAnt, Finalize and Code Motion.

Λ-Insertion phase and Renaming phase depend on SSU representation in original SSUPRE algorithm. We give up the SSU representation for the heavy overhead of its construction. Note that the only necessary information for Λ-Insertion is the places of λs. So the versions of these variables and λs in SSU form are unnecessary information for SSUPRE.

In other words, the effect of SSU representation can be replaced by that of iterated post-dominator frontier (PDF^+)of uses. Then we can save the overhead of SSU construction, which is one of the reasons for our more efficient solution.

Another reason is that our solution is fully sparse while the original Renaming phase of SSUPRE algorithm is not. We will explain below.

Occurrence Collection phase that collects real store occurrences is before all the phases. This phase creates one worklist for each store (i.e. lexically unique) candidate. The optimizer executes the next six phases for each worklist to eliminate the partial redundancies. The entire process is shown in Figure 6.

For a given worklist, there are many alias stores in the whole program. So it must traverse every statements of the whole program for each worklist during the Renaming phase of original SSUPRE since alias stores block the code motion. In this regard, there is no duality between SSAPRE and SSUPRE because alias loads have no effect in SSAPRE. For this reason, the SSUPRE Renaming that is the dual algorithm of sparse SSAPRE Renaming is dense. Traversals for rename phase through the whole program are inefficient.

We propose a fully sparse solution here and avoid too many traverses of the program. We added the collection of alias stores and alias loads in Occurrence Collection phase. For each worklist (store candidate), corresponding

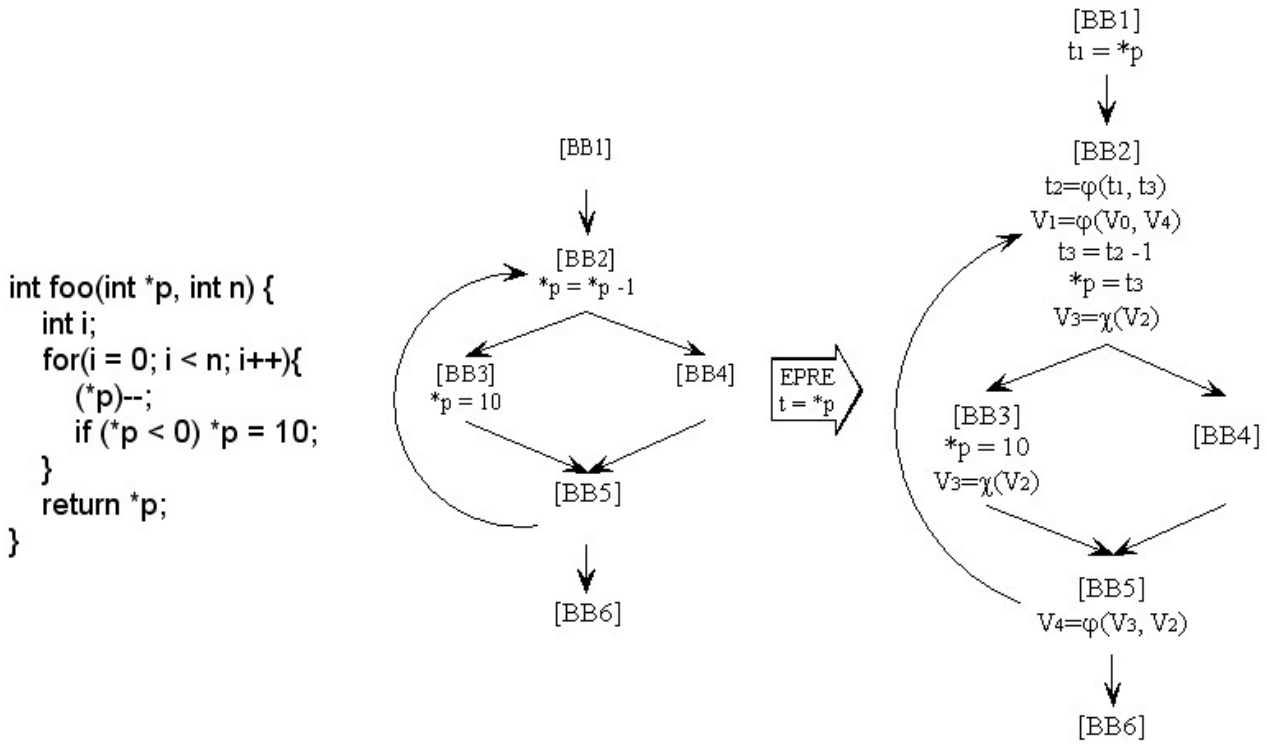


Figure 7. Sample code of C language, its original control flow graph and its SSA form after EPRE.

alias stores and alias loads appear as special occurrences. Then the other phases are performed fully sparsely. The complexity of the original SSUPRE and our SSUPRE is discussed in detail in section 4.

We define 6 kinds of occurrences: real occurrences, Λ occurrences, Λ operand occurrences, alias use occurrences, alias definition occurrences and entry occurrence.

Real occurrences indicate scalar stores and indirect stores in the original program, which are the real candidates of store PRE. In general, the left hand sides of evaluation statements are real occurrences.

Alias use occurrences are uses of variables that are alias with real occurrences while alias definition occurrences are definitions of variables that are alias with real occurrences. Note that we regard the use occurrence of a variable as the alias use occurrence of the worklist of itself. But alias definition occurrences is different from real store occurrences in SSUPRE. Real occurrences, alias use occurrences and alias definition occurrences are collected during Occurrence Collection phase that is discussed in subsection 3.1.

Λ occurrences are inserted during $\Lambda - Insertion$ phase and their operands are called Λ operand occurrences. In renaming phase, Λ operand occurrences are regarded as the successors of the basic block of Λ occurrences. We will discuss how to insert them in subsection 3.2 and discuss their versioning in subsection 3.3

Entry occurrence is a virtual occurrence that is before the first statement of the whole program. It is just used to compute upsafety conveniently. We will use it in the subsection 3.3.

We will use the a sample shown as Figure 7 to illustrate the process of our SSUPRE algorithm. The left part of Figure 7 is the sample code of C language and the middle part shows the control flow graph of the code. After EPRE optimization, we obtains the right part of Figure 7 that is represented by SSA form. In that graph, χ means a variable or memory address may be defined and the symbol V represent virtual variable corresponding to the memory address somewhere. Above concepts are defined by [10] in detail and we will use them in subsection 3.7.

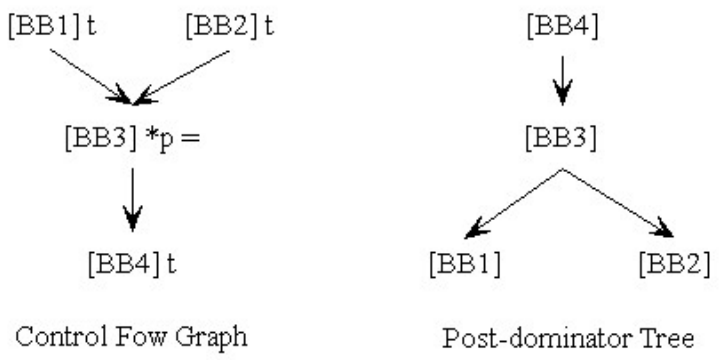


Figure 8. The case which cannot be traversed by one pass.

3.1 Occurrence Collection

The first task of SSUPRE is to collect occurrences SSUPRE algorithm needs. In this phase, real occurrences, alias use occurrences, alias definition occurrences and the entry occurrence are collected in two passes through the whole program.

We use the method *worklist-driven* that puts all real occurrences for the same symbol into the same worklist, i.e. one worklist is for the lexical identified symbol. [5] In Figure 7, $*p$ and t are the candidates for SPRE. So the optimizer creates two worklists for $*p$ and t. The worklist for $*p$ has two real occurrences in BB2 and BB3.

Alias use occurrences and alias definition occurrences are collected in the first phase for the sparseness of the algorithm. Otherwise, the optimizer will scan the whole program for each worklist in Λ-Insertion phase and Renaming phase. Suppose that $*p$ is alias with t so that the worklist for $*p$ has two alias use occurrences included the use occurrence of $*p$ and two alias definition occurrences.

The collection requires 2 passes of the post-dominator tree of the program in preorder.

The first pass only collects real occurrences and the entry occurrence. When encounter a scalar store or an indirect store, we create one worklist for the store if the store appears for the first time. Otherwise, the store is added into the existent corresponding worklist.

The second pass collects alias use occurrences and alias definition occurrences of the real occurrences. For a use occurrence A of a variable x that is alias with variable y whose worklist is S, append A into the list of alias use occurrences of S. For a definition occurrence B of a variable x alias with variable y whose worklist is T, append B into the list of alias definition occurrences of T.

Note that the alias uses/definitions may appear before or after the first appearance of the related real occurrence in the program. The worklist of a store may be created after meeting some of its alias uses/definitions so that one pass of the program is not enough. In Figure 8, suppose $*p$ is alias with t. The use occurrences of t appear at both before and after the store occurrence of $*p$ in the view of the post-dominator tree.

3.2 Λ-Insertion

We know that Φs are placed under 2 situations in SSAPRE algorithm [3]: iterated dominance frontiers (DF^+) of experssions and ϕs of variables contained in expressions (SSAPRE uses SSA [6] representation as its input form). As the dual algorithm of SSAPRE, SSUPRE should insert Λ at iterated post-dominance frontiers (PDF^+) of stores and λs of variables stored if SSU representation exists.

SSAPRE	SSUPRE
DF^+ of expressions	PDF^+ of stores
ϕ of variables	λ of varibles

Table 1. Duality of SSAPRE Phi-Insertion and SSUPRE Λ-Insertion

Observe that the construction of SSU is to identify the placement of λ variables which are the PDF^+ of use occurrences of the variables. So, without SSU, the places Λ should be inserted are the PDF^+ of use and definition occurrences.

Alias load/store processing is another problem. For a scalar store occurrence, we should consider 4 kinds of related occurrences as Table 2 shows.

Places	Insert Λ decision
PDF^+ of load	Insert Λ
PDF^+ of alias load	Insert Λ
PDF^+ of store	Insert Λ
PDF^+ of alias store	Insert Λ

Table 2. Λ-Insertion for scalar store x.

There are two kinds of stores, scalar stores and indirect stores. For an indirect store occurrence such as $*p$, there are 8 kinds of related occurrences as Table 3 shows.

Places	Insert Λ decision
PDF^+ of load *p	Insert Λ
PDF^+ of alias load *q	Insert Λ
PDF^+ of load p	Not insert Λ
PDF^+ of load *r	Not insert Λ
PDF^+ of store *p	Insert Λ
PDF^+ of store p	Insert Λ
PDF^+ of store *r	Insert Λ
PDF^+ of alias store *q	Insert Λ

Table 3. Λ-Insertion for indirect store *p. *q is alias with *p. *r is alias with p

For the worklist of $*p$ in Figure 7, Λs should be placed at the end of BB2 and BB5. The result after Λ-Insertion is shown as the left part of Figure 10.

3.3 Renaming

The renaming algorithm is like the reverse of SSAPRE renaming algorithm. The UpSafety properties of the Λ occurrences are set TRUE initially.

We traverse the post-dominance tree in depth first order and maintains an occurrence stack. As subsection 3.1 mentioned, there are 6 kinds of occurrences: Λ occurrences, real occurrences, Λ operand occurrences, alias use occurrences, alias definition occurrences and entry occurrence. When it encounters,

1. Λ occurrence, create a new version and push this Λ occurrence into the occurrence stack;
2. Real occurrence, if stack is empty or the top of the occurrence stack is $\perp$, create a new version and push it into the stack; otherwise, set the version of the real occurrence as the top of the occurrence stack;
3. Λ operand occurrences, set the version of the occurrence as the top of the occurrence stack (including $\perp$).
4. Alias use occurrence, insert $\perp$ into stack because that alias use occurrence kills use-definition chain. If the current top of occurrence stack is a Λ occurrence, this Λ occurrence is not UpSafe following the definition of UpSafety.
5. Alias definition occurrence, insert $\perp$ into stack because that alias definition occurrence blocks code motion in SSUPRE. For the conservative consideration, the top of the occurrence stack is not UpSafe if it is a Λ occurrence.

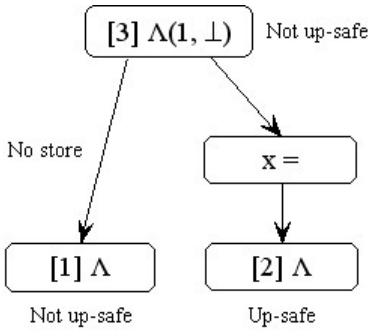


Figure 9. UpSafety Propagation.

6. Entry occurrence. Entry occurrence is for initialization of UpSafety conveniently. When we see an entry occurrence and the top of the occurrence is a Λ occurrence, the Λ occurrence must be not UpSafe following the definition of UpSafety.

After renaming, each Λ occurrence, real occurrence and Λ operand occurrence have their versions. And some of Λ occurrences are set not UpSafe. The result after Renaming is shown as the middle part of Figure 10.

3.4 UpSafety

The initial values of UpSafety of Λ occurrences are set in Renaming phase. This phase forward propagates FALSE of UpSafety via the edges of FRG. It is the exact reverse of SSAPRE DownSafety phase.

The UpSafe property of a Λ occurrence P is FALSE iff,

1. A result of P appears as an operand of a Λ occurrence Q;
2. Q is not up-safe.

In Figure 9, there is no store between Λ_1 and Λ_3 that is not up-safe. And a store that keeps Λ_2 up-safe is between Λ_2 and Λ_3. So Λ_1 is not up-safe and Λ_2 is up-safe.

3.5 WillBeAnt

WillBeAnt phase is exact reverse of SSAPRE WillBeAvail phase that includes two passes.

The first pass computes property can_be_ant which indicates whether a Λ can be anticipated. Initially, can_be_ants of the Λs that are not up-safe and have at least one ⊥ are set to false. False value is propagated to the not up-safe Λs via the FRG edges that have not real occurrences.

The second pass computes property earlier for each Λ, which is TRUE if the insertion can be *earlier*. It initializes to all can_be_ant Λ to TRUE. FALSE values are propagated from real occurrences via FRG edges.

The will_be_ant can be computed,

$$will\_be\_ant = can\_be\_ant \wedge !ealier \tag{1}$$

Only the will_be_ant Λs are the candidates for code motion phase that performs insertions and deletions. The result after Renaming is shown as the right part of Figure 10.

3.6 Finalize

Finalization identifies fully redundant stores and points out the locations that insertions are to be performed. A preorder traversal of post-dominantor tree is enough. An insertion must satisfies,

1. the Λ satisfies will_be_ant; and
2. the operand is ⊥; or has_real_occ is FALSE for the operand and the operand is defined by a Λ that is not will_be_ant.

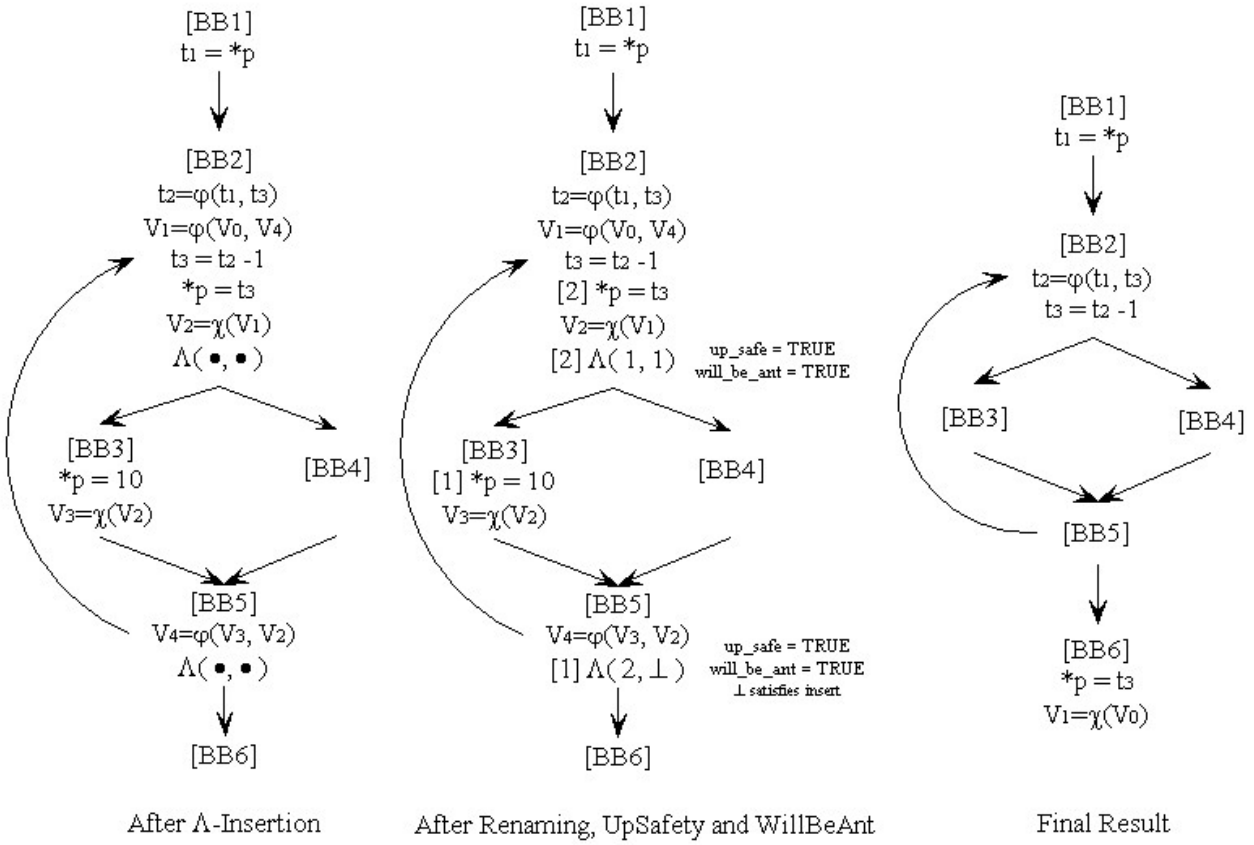


Figure 10. Results of SSUPRE phases.

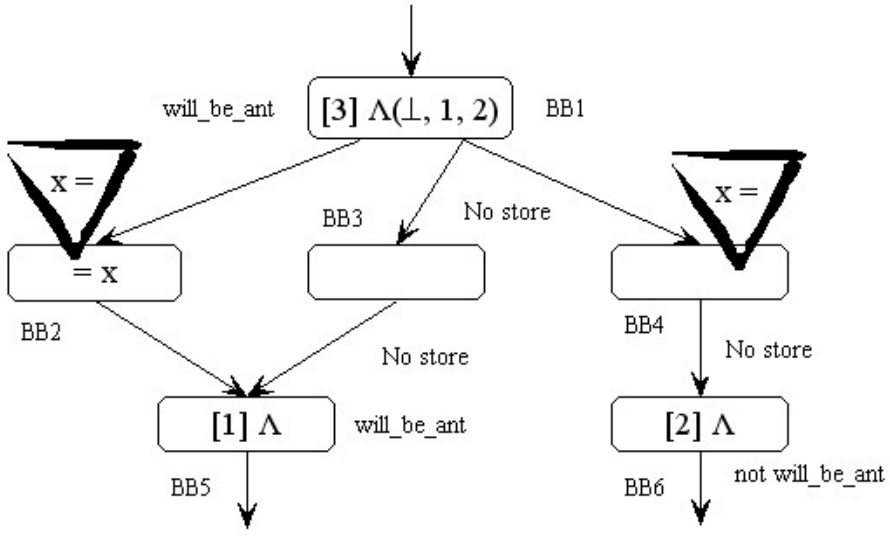


Figure 11. A subgraph of store insertion.

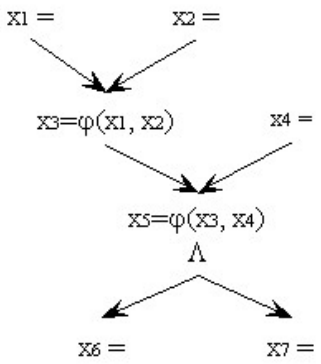


Figure 12. Λ occurrence and its redundancies.

These two conditions are shown as Figure 11. Suppose that Λ_1 and Λ_3 are will_be_ant and Λ_2 is not will_be_ant. BB2 has a use that kills Λ_1 and the corresponding operand of Λ_3 is $\perp$. So a store of x should be inserted at the entry of BB2. BB3 does not need insertion because Λ_1 is will_be_ant while BB4 need insertion because Λ_2 is not will_be_ant.

For an will_be_ant Λ occurrence, the optimizer must find all store occurrences that can reach the Λ occurrence without blocked by use. Such store occurrences are the redundancies of this Λ occurrence, which should be deleted. The Λ in Figure 12 is up-safe and fully anticipated. x_1, x_2 and x_4 are the redundancies of this Λ. It is a many-one relationship that is hard to analyze efficiently. We only need to record the closest occurrence (real occurrence or SSA ϕ occurrence) to the Λ in the view of dominator tree. In Figure 12, the Λ is associated with x_5. During the deletion of the redundancies, we simply traverse bottom-up from x_5 along the SSA use-definition chain to find all the redundancies belong to the Λ.

3.7 Code Motion

Code Motion phase performs actual insertions and deletions.

For an insertion of a Λ operand, insert a store statement at the entry of the corresponding successor of the Λ. The left hand side of the statement is a variable or memory address and the right hand of the statement must be a pseudo-register. The real occurrence $x \leftarrow < expr >$ that has the same version as the Λ is forced to be separated to 2 statements: $r \leftarrow < expr >$ and $x \leftarrow r$, which r is a pseudo-register. Figure 13 shows the process of making right hand expression to pseudo-register.

After insertions that make the Λs fully anticipated, the redundancies that belong to the will_be_ant Λs should be deleted. As we mentioned in subsection 3.6, each will_be_ant Λ has a store occurrence (real occurrence or SSA ϕ occurrence). We simply traverse bottom-up from the store occurrence until a real occurrence is encountered. In other words, the bottom-up traversal ends at real occurrences and these real occurrences are the redundancies of the Λ. In Figure 12, x_1, x_2 and x_4 are the endpoints of the trace tree. They should be deleted.

Finally, we rename the occurrences of the pseudo-register which is used to store the right hand side of the redundancies since the output of SSUPRE should be SSA representation.

The final result of our sample in Figure 7 is shown as the right part of Figure 10. The store statements of $*p$ are completely removed in the loop and the new store statement is inserted into the entry of BB6.

4. Analysis

Let v be the number of nodes and e be the number for edges in SSU graph. The complexity of Λ-Insertion ($O(v^2)$), UpSafety ($O(v+e)$), WillBeAnt ($O(v+e)$), Finalize ($O(v+e)$) and Code Motion ($O(v+e)$) of Our SSURPE is as same as that of the original SSUPRE.

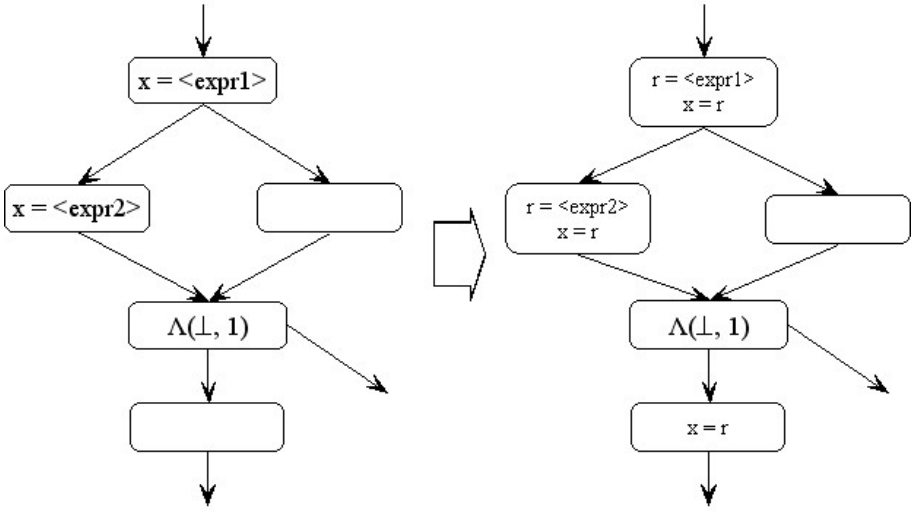


Figure 13. Right hand side forced to pseudo-register during store insertion.

The original SSUPRE creates SSU representation that is $O(v^2)$ and *Our SSUPRE saves the load of SSU construction.*

In Renaming phase, original SSUPRE must traverse the whole program for each worklist in theory. It will be many passes. So the original implementation traverses the whole program by only one pass to rename the worklists and get parallel effect. For each statement in the pass, it finds the associated worklist that has its own renaming stack. Then rename the occurrences of this worklist on its renaming stack. Such method assumes that the store candidates of worklists are not alias with each other. If the candidate variable of worklist A is alias with the candidate variable of worklist B, the insertion of worklist A in Code Motion phase is the alias store that can break the renaming result of worklist B. *That means the one-pass parallel renaming is only for scalar stores that have no alias with each other. For indirect stores, renaming for worklists must be performed one by one.*

Our sparse solution is to collect alias use occurrences and alias definition occurrences at first. Then renaming phase does not need to traverse the statements one by one as the original SSUPRE renaming phase. It just processes the pre-collected alias occurrences. Note that the insertions of a worklist should be added into the alias definition occurrence lists of the unsettled worklists after the worklist finished.

The time complexity of original theoretical SSUPRE renaming is $O(mn)$, where m is the number of worklists and n is the size of program. The original parallel renaming reduce it to $O(n)$. The time complexity our new SSUPRE renaming is $O(mc)$ that is a little bit larger than $O(n)$ in general, where c is the number of occurrences. Obviously, the parallel method uses m times spaces of the sequential methods.

5. Measurements

Our benchmark is a part of SPECCPU 2000 and SPECCPU 2006 running on 1.6GHz Itanium II processor. We divide the test cases into two sets: C program and C++ program. [1]

First, we summarize the numbers of compile time original stores, inserted stores and deleted stores in some cases of SPECCPU 2000 and SPECCPU 2006 in Table 4. As expected, indirect stores in C++ programs are much more than indirect stores in C programs generally.

The compile time of Store PRE phase for some cases of SPECCPU 2000 and SPECCPU 2006 is shown in Table 5. We run the original Store PRE phase for reference. Although the compile time of Store PRE phase of our new implementation is 3.4 times of that of the original implementation which omits elimination of partial

[1] We will perform more experiments with more C and C++ programs in near future.

Case	Type	Scalar Store	Indirect Store	Total Insertion	Total Deletion	Scalar Store Insertion	Scalar Store Deletion	Indirect Store Insertion	Indirect Store Deletion
164.gzip	C	3361	230	234	438	230	430	4	8
175.vpr	C	9098	839	672	920	657	898	15	22
197.parser	C	7246	714	512	709	502	672	10	37
177.mesa	C	35939	7032	1982	2147	1881	2036	101	111
183.equake	C	1478	127	69	194	65	188	4	6
252.eon	C++	43941	23501	637	4337	560	1104	77	3233
483.xalancbmk	C++	444932	72227	12785	30565	11414	25776	888	4789
450.soplex	C++	39337	6293	992	1392	971	1214	21	178
453.povray	C++	79404	14399	3150	4791	3007	4421	143	370

Table 4. Number of compile time stores in some cases of SPECCPU 2000 and SPECCPU 2006.

Case	Type	Original SPRE	New SPRE	Ratio
164.gzip	C	0.029937	0.072098	2.408324147
175.vpr	C	0.063038	0.220402	3.496335544
197.parser	C	0.061984	0.194008	3.129969024
177.mesa	C	0.277901	1.21475	4.371160953
183.equake	C	0.015705	0.082862	5.276154091
252.eon	C++	2.0579	4.81357	2.339068954
483.xalancbmk	C++	5.27897	12.3048	2.330909249
450.soplex	C++	0.347467	0.994928	2.863374076
453.povray	C++	0.810145	5.36758	6.625455937
Geom. mean				3.413190368

Table 5. Compile time in seconds of original SPRE and our new SPRE for some cases of SPECCPU 2000 and SPECCPU 2006.

indirect stores, our implementation is a practical and acceptable solution since the new phase only add seconds in total compilation time.

6. Conclusion and Future Work

Our new implementation of SSUPRE algorithm uses SSA representation as its input and output form without SSU representation. That causes the lighter overhead so that the optimizer is able to process elimination of indirect stores. The effect of this implementation for C++ programs is much better because C++ programs have more indirect stores.

We will continue our research on some detail issues about SSUPRE. The first is demand-driven Λ-Insertion without SSU form. It seems that demand-driven Λ-Insertion for SSUPRE has room to be improved. The second one is store sinking that can reduce the code size. The same store statements in the both branches can be sunk to the join point. That is much like the dual problem of expression hoist with some subtle differences. We will present the solution to these issues in our future works.

References

[1] Morel, E., Renvoise, C. Global Optimization by Suppression of Partial Redundancies. Communication of the ACM. 1979, Vol.22, No.2, pp 103.

[2] Open64 Compiler Project. http://www.open64.net.

[3] Chow, F., Chan, S., Kennedy, R., Liu, S., Lo, R., Tu, P. A New Algorithm for Partial Redundancy Elimination based on SSA Form. ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation. 1997.

[4] Ananian, C. S. The static single information form. Master's thesis, MIT, September 1999. Tech. Report MIT-LCS-TR-801.

[5] Lo, R., Chow, F., Kennedy, R., Liu, S., Tu, P. Register Promotion by Sparse Partial Redundancy Elimination of Loads and Stores. ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation. 1998.

[6] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., Zadeck, F. K. An Efficient Method of Computing Static Single Assignment Form. The Sixteenth ACM Symposium on Principles of Programming Languages. 1989.

[7] Knoop, J., Ruthing, O., Steffen, B. Lazy Code Motion.ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation. 1992.

[8] Kennedy, R., Chan, S., Liu, S., Lo, R., Tu, P., Chow, F., Partial Redundancy Elimination in SSA Form. ACM Transactions on Programming Languages and Systems. Vol.21, No.3, pp 627, May 1999.

[9] Kennedy, K. Safety of Code Motion. International Journal of Computer Mathematics. 1972, Vol.3, pp 117.

[10] Chow, F., Chan, S., Liu, S., Lo, R., Streich, M. Effiective Representation of Aliases and Indirect Memory Operations in SSA Form. Compiler Construction, 6th International Conference, 1996.

[11] Johnson, R., Pearson, D., Pingali, K. The Program Structure Tree: Computing Control Regions in Linear Time. ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation. 1994.

[12] Sreedhar, V., Gao, G. A Linear Time Algorithm for Placing ϕ-nodes. Conference Record of the Eighteenth ACM Symposium on Principles of Programming Languages. 1995.